INTERNAL MAINTENANCE DOCUMENTATION

6000 SCOPE 3.2

M628a

SCOPE

TABLE OF CONTENTS

1.0	INTRODUCTION	1-1
2.0	CENTRAL MEMORY RESIDENT (CMR)	2-1
	2.1 CMR Pointer Area	2-4
	2.2 PP Communication Areas	2-9
	2.3 Control Point Area	2-11
	2.4 Central Processor Resident Program	2-16
	2.5 Equipment Status Table	2-16
	2.6 Channel Status Table	2-19
	2.7 Mass Storage I/O Processing	2-20
	2.8 FNT/FST	2-35
	2.9 Installation Area	2-37
	2.10 Dayfile Buffer Area	2-38
	2.11 Library Directory	2-39.1
	2.12 System Definitions	2-42
	2.13 PPTEXT Symbols	2-44
	2.14 PPTEXT Macros	2-74
3.0	PERIPHERAL PROCESSOR RESIDENT	3-1
	Introduction	3-1
	Pool Processor Structure	3-1
	The Resident	3-1
	3.1 Resident Routines	3-3
4.0	DEAD-START	4-0
	4.1 Introduction	4-1
	4.2 General Flow	4-3
	4.3 CM Allocation	4-11
	4.4 CONTROL	4-13
	4.5 IRCP	4-29
	4.6 Dead Start Common Decks	4-50
	4.7 Hardware Aspects of Dead Start	4-50
	4.8 Operator Communication Package (OPCOM)	4-59
	4.9 Dead Start Debugging Aids	4-65

5.0 MTR - SCOPE SYSTEM MONITOR 5-1
5.1 Basic Definitions 5-1
5.2 Main Loop and Main Control 5-4
5.3 CPU Request Processing 5-7
5.4 PPU Request Processing 5-9
5.5 Memory Allocation - Storage Move 5-21
5.6 CPU Assignment 5-22
5.7 PPU Assignment 5-23
6.0 SYSTEM DISPLAYS 6-1
6.1 Overlay Structure and Organization of Memory 6-3
6.2 Making One's Way Through the Listing 6-5
6.3 General Logic Flow and Main Loop Subroutines 6-5
6.4 Keyboard Input Processing and Command Execution 6-8
6.5 Adding a Command 6-13
6.6 The Displays (General) 6-22
6.7 LOV - The Overlay Loader 6-24
6.8 LDOV - Load Display Overlay 6-24
6.9 The Displays (Individual) 6-25
6.10 Some Notes on the Macros 6-32
6.11 Resident DSD Subroutines and Exits 6-33
6.12 Tables in DSD 6-35
6.13 Locations of Interest 6-35
6.14 The Macros in PPMAC 6-36
6.15 DIS 6-47
6.16 9DM 6-55
6.17 1MH 6-55
7.0 CIRCULAR INPUT/OUTPUT 7-0
7.1 CIO 7-1
7.2 4ES 7-8
7.3 6WM 7-13
8.0 JOB PROCESSING 8-1
8.1 1RA - Resource Allocator 8-2
8.2 1AJ - Job Advancement 8-5
8.3 1EJ - End-of-Job-Processor 8-11
8.4 1LT - Load Job from Tape 8-17

9.0 STACK PROCESSOR - SCOPE 3 MASS STORAGE I/O PACKAGE 9-1
9.1 Rationale 9-1
9.2 Overlay Structure 9-2
9.3 Stack Processor Initiation 9-4
9.4 Stack Processor Completion 9-4
9.5 Order Codes 9-5
9.6 Error Codes 9-7
9.7 System Interface 9-10
9.8 PP Memory Allocation 9-13
9.9 Narratives 9-17

10.0 CHECKPOINT/RESTART 10-1
10.1 CKP - Tape Checkpoint 10-1
10.2 CKP Subroutines 10-2
10.3 RESTART - Checkpointed Job 10-3
10.4 RST - Restore Control Point Area for RESTART 10-4
10.5 1RC - Reload Core for CKP or RST 10-4
10.6 CY1 - Reset FNT 10-5

11.0 FILE ACTION REQUESTS 11-0
11.1 OPEN 11-1
11.2 CLOSE 11-7
11.3 MULTIFILE FNT Entry 11-12

12.0 GENERAL PURPOSE SYSTEM LOADER (GPSL) - CP LOADER 12-0
12.1 Process Description 12-1
12.2 Interfaces 12-6
12.3 Process Flow Chart 12-13
12.4 LOAD, Initializing Loader 12-14
12.5 LDR, CM Loading and Program Relocation 12-18
12.6 LOADER, Bookkeeping and Program Linking/Delinking 12-31
12.7 CP LOADER 12-68

13.0 SYSTEM PERIPHERAL PROCESSOR ROUTINES 13-1
ACE Control Card Reader 13-2
APR Automatic Program Sequencer 13-4
CHK Check Output File 13-12
CTS COMMON File Processing 13-13
DMP Dump CM 13-16

13.0 SYSTEM PERIPHERAL PROCESSOR ROUTINES (Continued)

LDV Absolute Overlay Loader 13-28
LOC Load Octal Cards 13-30
MDI Move System Directory (EDITLIB) 13-31
MEM Process MEMORY Function 13-36
MSG Add Message to Dayfile 13-155
REQ REQUEST Card/Function Processing 13-37
RFL Request Field Length 13-42
SRB Enter Expanded Disk Address into Directory (EDITLIB) 13-43
XDQ PP Counterpart of XXXDMPQ 13-46
XRQ PP Counterpart of XXXRESQ 13-48
1BT Blank Label Tape and Disk 13-49
1DF Dump Dayfile 13-50
1MF Multifile Positioning 13-51
1MR OPEN Read/Alter Magnetic Tape File 13-52
1MW OPEN Write Magnetic Tape File 13-53
1PL Dummy Plot Program 13-54
1RI Roll-In 13-55
1RO Roll-Out 13-58
1TD Tape Dump 13-60
2BP Check Buffer Parameters 13-62
2DF Drop File(s) 13-64
2LP On-Line Printer Driver 13-65
2PC On-Line Punch Driver 13-88
2RC On-Line Card Read Driver 13-105
2TJ Translate Job Card 13-115
3RP Close Reel Processing 13-117
5DA Private Disk Packs 13-123
6PC Drop Permanent Mass Storage Files 13-154

14.0 SYSTEM CENTRAL MEMORY ROUTINES 14-1

14.1 BKSP 14-1
14.2 COMBINE 14-1.1
14.3 COMPARE 14-2
14.4 COPY 14-11

14.0 SYSTEM CENTRAL MEMORY ROUTINES (Continued)

	14.5 COPYBCD	14-12
	14.6 COPYBF	14-15
	14.7 COPYL	14-19
	14.8 COPYN	14-24
	14.9 COPYSBF	14-34
	14.10 CPC	14-35
	14.11 DMPECS	14-43
	14.12 EDITLIB	14-46
	14.13 EDITSYM	14-98
	14.14 IO	14-112
	14.15 IORANDM	14-118
	14.16 RETURN	14-125
	14.17 REWIND	14-126
	14.18 UNLOAD	14-126
	14.19 UPDATE	14-127
	14.20 XXXDMPQ	14-148
	14.21 XXXRESQ	14-151
	14.22 TRANSF	14-153
15.0	64/6600 COMPASS DEBUG AIDS	15-1
	15.1 SNAP	15-1
	15.2 TRACE	15-14
	15.3 DEBUG	15-35
	15.4 OVERLOG	15-54
	15.5 Debug Aid ERROR MESSAGES	15-57
16.0	UNIT RECORD I/O	16-1
	JANUS I/O Package	16-1
	1IQ	16-1
	1IR	16-22
	1IS/2IS	16-124
	1IU	16-126
17.0	TAPE I/O	17-0
	17.1 CIO - Circular Input/Output	17-1
	17.2 1MT - Long Record Stranger Tape Driver	17-2
	17.3 1PE - Tape Write Parity Error Recovery Routine	17-15
	17.4 1RS - S-Tape Read Driver	17-18

17.0 TAPE I/O (Continued)

17.5 1RT - Scope and X-Tape Read Driver 17-23
17.6 1WI - Tape Writer Driver 17-24
17.7 1WS - Tape Write Driver 17-28
17.8 1WX - Write External Tapes 17-36
17.9 2TB - Backward Positioning ½" Magnetic Tape 17-43
17.10 1TF - ½" Forward Tape Motion 17-44
17.11 4LB - Label Reading/Writing 17-46
17.12 4LC - Y-Tape Processor 17-50

18.0 PERMANENT FILES I 18-0

18.1 Introduction 18-1
18.2 Functions 18-8
18.3 PFM Main Line 18-12
18.4 Routine Details 18-14
18.5 PFC 18-14
18.6 PFA 18-22
18.7 PFP 18-28
18.8 PFE 18-29
18.9 Common Routines 18-39
18.10 System Interface 18-71
18.11 AUDIT 18-75
18.12 DPF - DUMPF 18-85
18.13 LPF - LOADPF 18-98

19.0 BNL ECS 19-1

19.1 Introduction 19-1
19.2 Direct User Access Capability 19-2
19.3 Allocatable Device Usage Capability 19-2
19.4 System Symbols for BNL ECS 19-5
19.5 CMR 19-7
19.6 MTR 19-14
19.7 Dead-Start 19-15
19.8 Stack Processor 19-20
19.9 ECS Statistics 19-25
19.10 Other Routines 19-26
19.11 Related Topics 19-28
19.12 Dead-Start ECS Display 19-30
19.13 ECS Partition Table 19-31
19.14 Sample Listing 19-33

20.0 C. E. DIAGNOSTICS 20-0
20.1 C. E. Diagnostic Programs 20-1
20.2 C. E. Diagnostic Error File 20-8
21.0 SYSTEM FILES 21-1
21.1 Dayfile Processing 21-1

1.0 INTRODUCTION

The SCOPE Operating System for the CONTROL DATA 64/65/6600 computers provides supervisory control of program execution. It provides the user with easy access to and control of all the advanced capabilites of the 6000 computer systems, and is designed to ensure optimum performance in processing of jobs.

SCOPE is in constant control of all jobs, handling storage allocation, job scheduling, accounting, I/O control, and operator communication. When used on the dual processor 6500, the system allows automatic scheduling of dual central processors within the multi-programming environment.

This document is the Internal Maintenance Specification for SCOPE 3.2, and supercedes all previous IMS's. Comments and suggestions for improvement are solicited.

2.0 Central Memory Resident - CMR

The Central Memory Resident provides communication between the various PP's and contains all the information shared by the system.

Both upper and lower memory are used for system storage and both ends are of variable length at assembly and execution time.

CMR includes:

The CMR Pointer Area which contains various flags and pointers to tables contained elsewhere in CM.

The PP Communication Area which contains ten 8-word areas, one for each PP, through which PP's communicate with each other and monitor.

The Control Point Area which contains seven 200B word areas, one for each control point. This area contains the exchange package, job name, and information about the job which is running at that control point.

The CP Resident Area which contains two programs which run at control point zero (the storage move program, and the idle package); their exchange packages are also kept in the CP Resident area. If there is ECS within the system, the CP area contains also an ECS move program.

The Equipment Status Table which contains one entry for each device (allocatable or non-allocatable) attached to the system. Non-allocatable devices are those which may be assigned to a single control point, e.g., magnetic tape unit; allocatable devices may be used by many control points simultaneously.

The Channel Status Table which contains one word per channel. A channel can be reserved by a peripheral processor via a monitor request (M.RCH). It can be released directly by the peripheral processor without MTR intervention.

The File Name Table/File Status Table which is composed of as many as 3-word entries as there are files in the system. An FNT entry is set up at the time that a file is created; it is not accessible to the user. This table provides a linkage between the user program and all required I/O tables and functions.

The Attached Permanent File Table which contains a one word entry for every permanent file attached to a control point.

<u>The Sub-Directory Table</u> which contains a one byte entry per permanent file sub-directory {other than sub-directory zero}.

<u>The Record Block Reservation Area</u> which contains at least one Record Block Reservation {RBR} table for each mass storage unit known to the system. Each RBR contains 2048 bits in which each bit indicates whether the corresponding record block is available for assignment to a file.

<u>The Request Stack Area</u> which is actually composed of two tables; the Device Status Table {DST} and the request stack itself. The DST contains one 2-word entry for each mass storage controller known to the system; it is static and its content is defined at assembly time. The request stack contains entries which are requests for data transfers, device positioning, or logical operations on mass storage files. Each entry is two words in length. The table grows from high memory to low.

<u>The Program Sequencer Table</u> which contains a one word entry for each job running under the control of the program sequencer {APR see Chapter 13}.

<u>The Installation Area</u> which is reserved for installation use.

<u>The Dayfile Buffer Area</u> which contains eight File Environment Tables and eight buffers, one for the system dayfile and one for each of the seven control points.

<u>The ECS Buffers and Tables</u> {see Chapter 19}.

<u>The Library Directory</u> which is composed of three sections; the entry point table, containing 1-word entries; the program name table, containing 2-word entries; and the bodies of the CM resident programs. The directory may be expanded or contracted as programs are added or deleted or as programs residence is changed.

<u>The Record Block Table Area</u> {RBT} which is a collection of individual file chains, one for each file on a mass storage device currently recognized by the system. When a file is initiated, a single two-word entry is assigned to the file. Additional two-word entries are assigned to the file as needed. The entries for a given file are chained together into a threaded list. The RBT empty chain is a pool from which words may be extracted to construct file chains and to which words are returned when the chains are discarded. The RBT area starts at the high end of central memory and extends downward; it expands and contracts by 100B word blocks as files are created and released.

CMR

0	P.ZERO	POINTERS	
60, 70, ...	T.PPC 1, 2, ...	PP COMMUNICATION AREAS	
200, 400, ...	T.CPA 1, 2, ...	CONTROL POINT AREAS	
		CP RESIDENT PROGRAMS	
	T.EST	EQUIPMENT STATUS TABLE	L.EST
	T.CST	CHANNEL STATUS TABLE	L.CST
	T.FNT	FILE NAME/STATUS TABLE	L.FNT
	T.APF	APF	L.APF
<100000B	T.SDT	SDT	L.SDT
	T.RBR	RECORD BLOCK RESERVATION TABLES	N.RBRUSE
	T.DST / T.RQS	DEVICE STATUS TABLE AND REQUEST STACK	L.RQS
<200000B	T.SEQ		L.SEQ
	T.INS	INSTALLATION AREA	L.INS
	T.NOVA		L.NOVA
	T.DFB	DAYFILE BUFFERS	
	T.ECSBUF		L.ECSBUF
	T.ICEBUF		L.ICEBUF
	T.ECST		L.ECST
	T.ECSTAT		L.ECSTAT
	T.ECFLAW		L.ECFLAW
	T.LRD		L.LRD
	T.LIB	LIBRARY DIRECTORY	

2.1 CMR Pointer Area (Location 0-59 of CMR)

Locations are described below by name in order of their occurrence.

P.ZERO

Location 0 - Zero Word
Location 0 of central memory, known as P.ZERO, is preset by the Dead-Start loader (STL) to contain a full word of binary zeros; a peripheral processor may clear a 5-byte area in its memory by reading this location. The introduction of a non-zero quantity into P.ZERO would therefore destroy the system totally. Location 0 is also used as a stop instruction by the CM resident and idle program.

P.LIB

Location 1 - Library Pointer
The library pointer (P.LIB) is used to define the size of the resident library, including entry point and program name tables. Bytes 0 and 1 (C.DIRFWA) contain the right justified 18-bit first word address of the library directory. Bytes 2 and 3 contain the right justified 18-bit last word address plus one. Byte 4 contains a Dead-Start load flag; it must always by zero when a disk or recovery Dead-Start is attempted. Location 1 is also used as a stop instruction in a 6500 system by the program idling in the second central processor.

P.RBR/P.RBT

Location 2 - RBR/RBT Pointer
Record Block Reservation/Record Block Table pointer. P.RBR/P.RBT is the joint address of the Record Block Reservation table pointer word and the Record Block Table pointer. Bytes 0 and 1 (C.RBRAD) contain the right justified 18-bit first word address of the RBR table area. Byte 2 contains the RBT word pair ordinal of the first member of the RBT empty chain, or zero when the empty chain has no members. Byte 3 contains the current length/100B of the entire RBT area. Byte 4 contains the (last word address+1)/100B of central memory.

P.DFB

Location 3 - Dayfile Pointer
P.DFB is the address of the dayfile buffer pointer word; Byte 0 contains the CM address/10B of the dayfile buffer.

P.FNT

Location 4 - File Name Table Pointer
P.FNT is the address of the File Name Table pointer word. Byte 0 contains the 12-bit first word address; byte 1 contains the 12-bit last word address plus one.

P.SEQ — Location 4 - Program Sequencer Table Pointer
P.SEQ is the location of the Program Sequencer Table pointer word. Byte C.SEQ contains the table starting address/10B. Byte L.SEQ contains the length of the table.

P.HEC — Location 4 - Hardware Error Count Pointer
Byte C.HEC contains the hardware error count.

P.CST — Location 5 - Channel Status Table Pointer
P.CST is the address of the Channel Status Table pointer word. Byte C.CST contains the starting address of the table. Byte C.CSTL contains the table last word address+1.

P.EST — Location 5 - Equipment Status Table Pointer
P.EST is the address of the Equipment Status Table pointer word. Byte 0 contains the 12-bit first word address. Byte 1 contains the 12-bit last word address plus one.

P.NCP — Location 5
Byte C.CNP contains the number of available control points.

P.PFM1 — Location 6 - Permanent File Pointer (See Chapter 18)

P.PFM2 — Location 7 - Permanent File Pointer (See Chapter 18)

P.INS — Location 10 - Installation Pointer
P.INS is the address of a pointer word to an installation area.

P.ECST — Location 11 - ECS Tables Pointer
Contains 5 pointers to ECS tables. Each pointer is 12 bits in length.
Byte 0 points to ICEBOX I/O Buffer (T.ICEBUF)
Byte 1 points to ECS Statistics Table (T.ECSTAT)
Byte 2 points to ECS Flaw Table (T.ECFLAW)
Byte 3 points to Logical Record Definition Table (T.LRD)
Byte 4 points to ECS Information Table (T.ECST)

P.LECST — Location 12 - ECS Tables Lengths
Contains 5 bytes which specify the length of ECS tables in P.ECST. The length of each ECS table is contained in the byte which corresponds to the location of the table pointer (see P.ECST).

P.RQS — Location 13 - Request Stack Pointer
P.RQS is the address of the request stack area pointer word. Byte 0 (C.RQSCS) contains the stack entry word pair count - a count of the number of word pairs available for storage of stack entries,

CMR - POINTER AREA

Symbol						Word
P.ZERO	ZEROS					0
P.LIB	0 0 C.DIR FWA FWA OF DIRECTORY			LWA+1 OF DIRECTORY	DEAD START LOAD FLAG	1
P.RBR/P.RBT	C. RBRAD FWA OF RBR AREA		RBT ORDINAL OF EMPTY CHAIN	LENGTH/100$_8$ OF RBT AREA	(LWA+1)/100$_8$ OF CM	2
P.NOVA/P.DFB	FWA/8 OF DAYFILE BUFFER	C.NOVA T.NOVA/8	C.NOVAL L.NOVA			3
P.SEQ/P.FNT/P.HEC	FWA OF FNT	LWA+1 OF FNT	C.SEQ T.SEQ/8	C.SEQL L.SEQ	C.HEC HDWR ERROR CNT	4
P.CST/P.EST/P.NCP	FWA OF EST	LWA+1 OF EST	C.CST T.CST	C.CSTL LWA+1 OF CST	C.NCP N.CP	5
P.PFM1	C.SDTL N.SD	C.APFL L.APF	C.SDT FWA OF SDT	C.APF FWA APF	C.PFICH PF INTERLOCKS	6
P.PFM2	C.SDL N.ESD	C.RBTCL N.RBTC	C.RBTC1 POINTER TO RBTC	C.RBTC2	C.RBTC3 E.O.I	7
P.INS		RESERVED FOR INSTALLATIONS				10
P.ECST	C.ICEBUF T.ICEBUF/8	C.ECSTAT T.ECSTAT/8	C.ECFLAW T.ECFLAW/8	C.LRD T.LRD/8	C.ECST T.ECST/8	11
P.LECST	C.ICEBUF L.ICEBUF	C.ECSTAT L.ECSTAT	C.ECFLAW L.ECFLAW	C.LRD L.LRD	C.ECST L.ECST	12
P.RQS	STACK ENTRY WORD PAIR COUNT C.RQSCS		C.RQSFS FWA/2 OF REQ STACK	# OF DST ENTRIES	FWA/8 OF DST	13
						14
						15
						16
						17
USE W. SYMBOLS AS FOR CONTROL POINT	0000	0000	0000	0000	MACHINE FIELD LENGTH	20
	S Y S T E M					21
				0	C.CPECFL FL ECS /1000$_8$	22
	7 7 7 7		CPU IDLE TIME SECONDS		MILLISECONDS	23
						24
			ECS TIME C.P. ZERO (SECONDS)		MILLISECONDS	25
T.CPJOBN	JOB SEQUENCE NUMBER		JOB COUNT			26
T.JDATE	0 0 0 0 0 Y Y D D D					27
T.CLK	* H H . M M . S S *					30
T.SLAB 1/T.DATE	M M / D D / Y Y					31
T.SLAB 2 . . . T.SLAB 6	SCOPE VERSION 3.X					32 ... 36
T.MSP	RESERVED FOR CDC DEBUGGER				STEP FLAG	37
T.MSC	COUNT OF PP JOB QUEUE ENTRIES	SECONDS	MILLISECONDS/1000$_{10}$	MILLISECOND CLOCK		40
T.PPS 1 . . . T.PPS 0	CONTROL POINT ADDRESS	T.PPS1 T.PPS2 ... T.PPS0				41 ... 52
	CDC					53
	CDC					54
	CDC					55
T.CPT 1/T.UAS/T.STATCP	UNASSIGNED CM/100$_8$	UNASSIGNED ECS/1000$_8$	C.STATCP	CPU B STATUS	CPU A STATUS	56
T.ECSPAR			ECS FLAW TABLE FLAG	ECS PARITY FLAG	ECS PARITY ADDRESS/1000$_8$	57

computed by subtracting the device status entry count (byte 3) from one half the length of the stack. Byte 2 (C.RQSFS) contains the first word address/2 of the actual request stack. Byte 3 contains the value of N.DEVICE; i.e., the number of mass storage controllers known to the system. Byte 4 contains the first word address/10B of the Device Status Table entries; all DST entries appear at the beginning of the request stack area, followed immediately by the actual request stack.

<u>(W.CPSTAT)</u> Location 20 - Control Point Zero Status Word
Byte 2 contains the storage move flag, byte 3 contains 0, and byte 4 the machine field length.

<u>(W.CPJNAME)</u> Location 21
This location contains the display code constant SYSTEM.

<u>(W.CPECS)</u> Location 22
Byte 4 contains the length of ECS: ECSFL/1000B.

<u>(W.CPTIME)</u> Location 23
This location contains the accumulated CP idle time, i.e. time spent at control point zero. (It also includes time used to move storage.)

<u>T.JDATE</u> Location 27
This symbol designates the location of today's date in Julian format. The value is computed from the date entered by the operator at Dead-Start time.

<u>T.CLK</u> Location 30
This symbol designates the location in which the clock is kept, in the display coded form *HH.MM.SS*. T.CLK is updated by MTR and displayed on the top line of the left scope. The time will be time of day, if a TIME entry to DSD was made; otherwise, it will be the time since Dead-Start.

<u>T.DATE</u> Location 31
This symbol designates the location of today's date in the form entered by the operator at Dead-Start time. The date is displayed on the top line of the left scope.

<u>T.SLABx</u> (where x = 1 through 6) - Location 31 - 36
These symbols designate the locations containing the system label which is displayed on the top line of the left scope.

<u>T.MSP</u> Location 37
This symbol designates the location of the monitor step flag (byte 4) used for communication between DSD and MTR while the system is in step mode.

<u>T.MSC</u> Location 40
T.MSC contains the time to the millisecond since Dead-Start.
Byte 0 = the number of jobs in the PP Job Queue
Byte 1 = time in seconds (reset each 2**12 seconds)
Byte 2 = milliseconds since updating the seconds
Byte 3-4 = time in MSEC (reset each 2**24 milliscds)

<u>T.PPSx</u> (where x = 1,2,...9) - Locations 41 through 51
These symbols designate the locations of the PP status words (one per PPU). If the PP is assigned to a control point, byte 0 will contain the appropriate control point area address.

<u>T.PPS0</u> Location 52
This location is used by MTR as a CM scratch word.

<u>T.UAS</u> Location 56
This symbol designates the location of unassigned storage length. MTR keeps a tally of the size of the "holes" between control points in T.UAS.
Byte 0 Central memory UAS (words/100B)
Byte 1 ECS UAS (words/1000B)

<u>T.CPT1</u> Location 56
This symbol designates the location of the Central Processors status bytes. The status is the control point area address of the program using the CPU. (This address is 0000 for idle.)
Byte 3 Active control point address CPU B
Byte 4 Active control point address CPU A

<u>T.STATCP</u> Location 56
This location contains in byte C.STATCP the current status of CPU ON/OFF/DELEGATED. The six low order bits indicate the status of CPU A, the six high order bits the status of CPU B.
The status of a CPU can be:
77 non-existent CPU (6400 or 6600)
60 CPU OFF (6500 only)
44 CPU DELEGATED (6500 only)
40 CPU ON

<u>T.ECSPAR</u> Location 57
Word 57B of CMR contains in bytes:
0, 1 Unused
2 An ECS Flaw Table Flag which is set to a non-zero value if the ECS flaw table becomes full.

In this case MTR puts the system in step mode and a warning message is flashed at the bottom of the right screen.

3 An ECS parity flag which can assume values of 1 or 2. It is set when a parity error is detected during an ECS storage move, and indicates whether 1 or 2 control points shall have their error flags set to F.ERECP. The control point requesting the ECS will always have its error flag set. In the case of storage overlap and the occurrence of the parity error in the part of ECS which belongs to another control point, the flag will have the value 2.
After the flag is set, DSD will display a message stating that an ECS parity error has occurred and MTR will assign ECS only if it does not involve moving storage. These conditions shall prevail until the next Dead-Start.

4 This byte contains the address of the block in ECS/1000B in which the parity error occurred. DSD will display this value as above.

2.2 PP Communication Areas (Locations 60B - 177B)

The PP Communication Area contains ten 8-word areas, one for each PP, through which PP's communicate with each other.

T.PPCx (x=1 - 9) - (60, 70, 100, 120, 130, 140, 150, 160)
The T.PPCx symbols represent the first word addressed of communication areas for PP's 1 through 9, respectively. The monitor communication area (T.PPC10) is not used.

W.PPIR (0)
W.PPIR is the relative location of the PP input register within a PP communication area.

W.PPOR (1)
W.PPOR is the relative location of the PP output register within a PP communication area.

W.PPMESx (x = 1 through 6)
W.PPMESx are the relative locations of the six words of the PP message buffer within a PP communication area.

Each peripheral processor contains pointers to its Input Register, Output Register, and Message Buffer in peripheral processor memory locations 75, 76, and 77, respectively. The communication areas are used to provide a means of communication between MTR and peripheral processor programs. When a peripheral processor is idle, its resident program continuously scans its Input Register. When MTR has a task for that processor, it sets the name of the appropriate routine in the Input Register of the idle processor, which, when it recognizes the request, loads the routine and executes it. MTR regularly scans the Output Register of each active peripheral processor. When a peripheral processor requires MTR assistance (such as, for example, reserving a data channel), it places a code in its Output Register. MTR detects the request during its scan of the output registers and processes it. When the request has been processed, MTR clears the requesting processor's Output Register; this informs the requesting processor that the request has been processed.

The six-word Message Buffer is used to pass parameters and messages between MTR and the peripheral processor resident programs.

PP COMMUNICATION AREA

W.PPIR	PROGRAM NAME OR 0	0 (IC.CPNUM)	CP#		0
W.PPOR					1
W.PPMES 1					2
W.PPMES 2					3
W.PPMES 3					4
W.PPMES 4					5
W.PPMES 5					6
W.PPMES 6					7

EXCHANGE PACKAGE

0		P	A0	B0	0
1		CMRA	A1	B1	1
2		CM FL	A2	B2	2
3		EM	A3	B3	3
4	ECS RA		A4	B4	4
5	ECS FL		A5	B5	5
6		MA	A6	B6	6
7			A7	B7	7
10	X0				8
11	X1				9
12	X2				10
13	X3				11
14	X4				12
15	X5				13
16	X6				14
17	X7				15

2.3 Control Point Area (Locations 200_8 - 1777_8)

The control Point Areas contain seven 200_8 word areas, one for each control point. The first 20_8 words of a control point area contain the exchange jump package for the central processor program which may be associated with this control point, as follows:

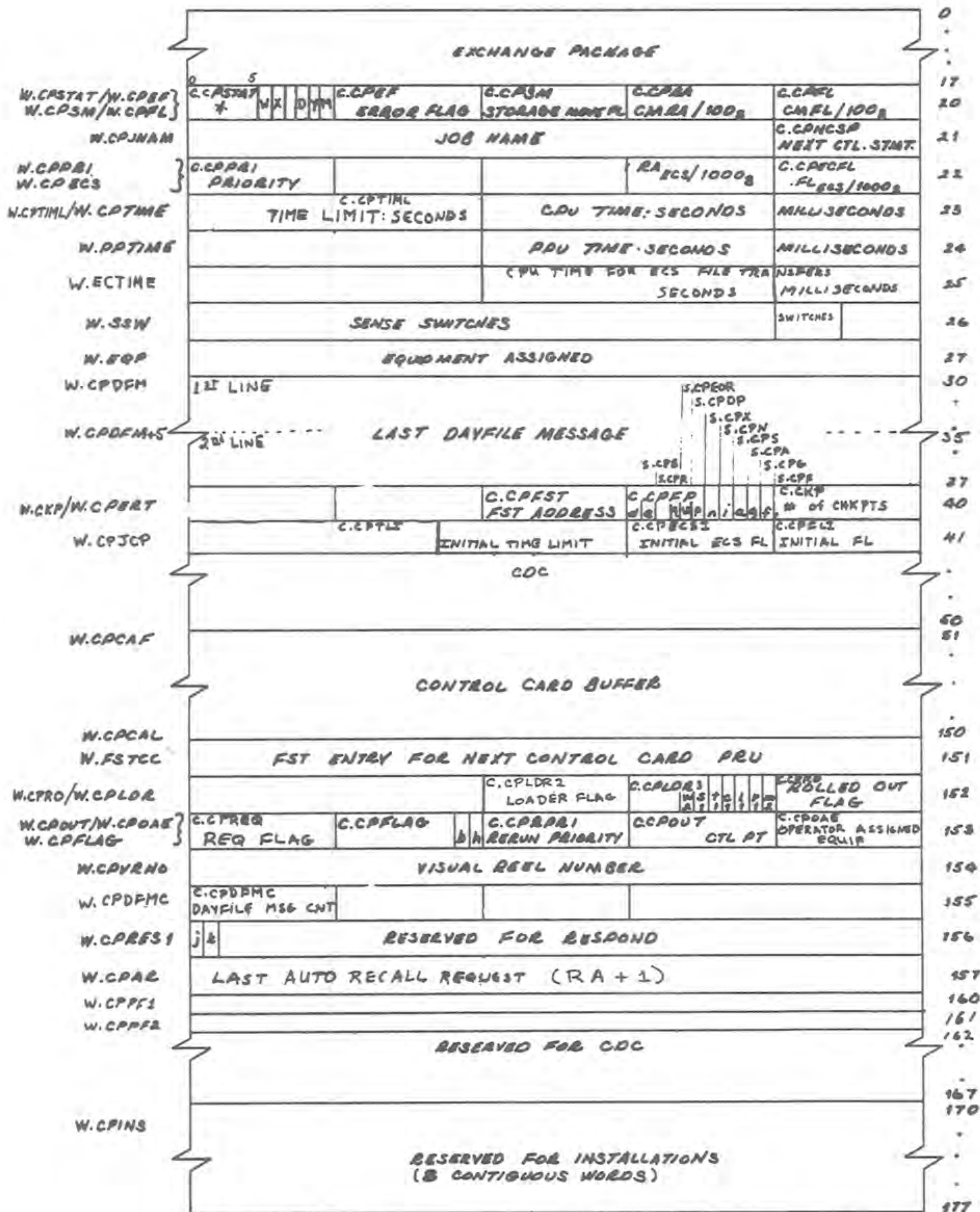
Word	bits 0-17	bits 18-35	bits 36-53
0		A0 (address registers)	Program Address (P)
1	B1 (increment register)	A1	Reference Address (CMRA)
2	B2	A2	Field Length (CMFL)
3	B3	A3	Exit Mode (EM)
4	B4	A4	RA-ECS (bits 36-59)
5	B5	A5	FL-ECS (bits 36-59)
6	B6	A6	Monitor address
7	B7	A7	
10	X0		
11	X1		
12	X2		
13	X3		
14	X4		
15	X5		
16	X6		
17	X7		

Words 20-152 of Control Point area

Word Mnemonic		Bytes Number	Item
W.CPSTAT	20	0	C.CPSTAT status of control point (see Chapter 5).
W.CPEF		1	C.CPEF error flag. If no error, C.CPEF=0. If error, C.CPEF may contain one of the values defined by F.x symbols.

Word Mnemonic		Bytes Number	Item
W.CPSM	20	2	C.CPSM Storage Move Flag. Mtr sets this to non-zero when storage attached to a control point is to be moved; all PP's operating at this control point should pause if C.CPSM $\neq$ 0.
		3	C.CPRA. RA/100B of control point.
W.CPFL	20	4	C.CPFL. CMFL/100B assigned to Control Point.
W.CPJNAM	21	0-3	Job Name (7 characters)
		4	C.CPNCSP. Pointer to next control statement.
W.CPPRI	22	0	C.CPPRI. Job Priority.
W.CPECS	22	3	RA/1000B of ECS words assigned to the control point.
		4	C.CPECFL. FL/1000B of ECS words assigned to the control point.
W.CPTIML	23	0-1	C.CPTIME (1) - CP time limit.
W.CPTIME	23	2-3	CPU time (sec.)
		4	CPU time (milliseconds)
W.PPTIME	24	1	Loader flag
		2-3	PPU time (sec.) accumulated by the job.
		4	PPU time (milliseconds)
W.ECTIME	25	2-3	CPU time used for ECS transfers (sec.)
		4	CPU time used for ECS transfers (milliseconds)
W.SSW	26	0-4	Sence switch settings
W.EQP	27	0-4	Current equipment assignment. Bits 59 through 0 correspond to equipment 01 through 74B respectively.
W.CPDFM	30-37		Last dayfile message words 30-34 = 1st display line words 35-57 = second display line
W.CPERT	40	0	C.CPPWCT - Number of CM words removed from current PRU in control card record.
		1	C.CPCCST - Status of last read of control card record of input file.
		2	C.CPFST - Pointer to FST of input file
		3	C.CPFP - Various flags (see figure).

CONTROL POINT AREA



a. RERUN BIT 0=return 1=no return
b. REBUN PRIORITY FLAG
c. CHANGE DUMP
d. EXPORT/IMPORT BIT
e. RESPOND BIT
f. REPROCESS BIT
g. ABORT BIT FOR EXIT
h. CLEAR FLAG
i. SEQUENCER FLAG
j RESPOND IDLE FLAG
p exit card encountered
q control card end of record

k RESPOND CONTINUE FLAG
l LABELED DUMP
m 00 - NO MAP; 01 FULL MAP; 10 PARTIAL MAP
r REDUCE FLAG
s SNAP
t TRACE
w LOADER 0=OLD; 1 = CP LOADER
n CHECKPOINT BIT, SET IF A VALID CHECKPOINT HAS BEEN TAKEN
u PRIVATE DISK PACK FLAG

* C.CPSTAT: W = wait D = delegated M: SUSPENDED FOR MOVE
X = recall Y = automatic recall

Word Mnemonic		Bytes Number	Item
W.CKP	40	4	C.CKP number of checkpoints
W.CPJCP	41	1-2	C.CPTLI initial job time limit
		3	C.CPECSI initial ECS FL
		4	C.CPFLI initial CM FL
W.CPCAF	51	0-4 first	Control card buffer first. Contains relative first word address within control point area of the 100B word buffer containing the current control card PRU.
W.CPCAL	150	0-4	Control card buffer last. Relative last word address within control point area of a 100B word buffer containing the current control card PRU.
W.FSTCC	151	0-4	FST entry (FNT word 2) for the job input file. Designates the position on the device of next PRU of control cards.
W.CPLDR	152	2	C.CPLAI loader already-in flag
		3	C.CPLDR3 loader map option
W.CPRO	152	4	C.CPRO roll-out flag
W.CPFLAG	153	1	C.CPFLAG - contains RERUN flags
		2	C.CPRPRI - designates the RERUN priority
W.CPOUT	153	3	C.CPOUT - flags to DSD that the job at this control point is an OUTPUT file.
W.CPOAE	153	4	C.CPOAE - designates operator assigned equipment
W.CPVRNO	154		Visual reel number (transmitted from console by the operator to the tape labelling routine)
W.CPDFMC	155	0	C.CPDFMC - dayfile message count
W.CPRES1	156	0-4	Used by RESPOND
W.CPAR	157	0-4	Auto-recall pointer. This pointer contains in the low order 77 bits a relative address of a word within the program field length; the program remains in recall until the low order bit of that word becomes one indicating operation completed.

Word Mnemonic		Bytes Number	Item
W.CPPF1	160	0-4	Used by permanent file manager
W.CPPF2	161	0-4	Used by permanent file manager
W.CPINS	170-177		Reserved for installations.

2.4 Central Processor Resident Program

The CP resident area contains programs which run at control point N.CP+1 with a priority of 7777B. They all share the same exchange package, located at (N.CP+1)*200B, where

```
 CM RA = 0       CM FL = 377 777B
ECS RA = 0      ECS FL = 10 000 000B
```

The programs are initiated by MTR, following a storage or a M.ICE request (see Chapter 5). Only one program is initiated at a time; it completes its execution by jumping to location zero.

Currently MTR recognizes 3 CP resident programs:

1 - CM storage move program
2 - ECS storage move program
3 - ECS transfer program

The CP resident area may also contain a central exchange jump protection program and its own exchange package. This program (only assembled if IP.XJ ≠ 0) is initiated by any user program executing a central exchange jump. It will simulate an error mode 10 before jumping back to location zero of the user program.

2.5 Equipment Status Table

The Equipment Status Table contains one (one-word) entry for each device (allocatable or non-allocatable) attached to the system. Non-allocatable devices are those which may be assigned to a single control point, e.g. magnetic tape unit; allocatable devices may be used by many control points simultaneously.

The table's base address is provided to the system by T.EST in byte zero of the EST pointer in CM location 5. Byte 1 of this pointer provides the last word address plus one, i.e. the sum of the contents of byte 0 and parameter L.EST designating the length of the table, which may vary from the number of entries to 100B.

In each entry of the table, bits 12-22 contain a hardware mnemonic, a two letter code, indicating a device. The first letter is entered in bits 18-22, the second in bits 12-17. For allocatable devices, a "one" in bit 54 indicates that the device is system resident.

2.5.1 The EDST Macro

The EDST macro is used in CMR to assemble an Equipment Status Table (EST) entry for each mass storage unit, and also generates a Device Status Table (DST) entry for each mass storage controller. The macro call is:

```
name    EDST   type,chan1,chan2,contr,unit,sysres,driver,onoff
```

name — is an identifier for the unit, one to six characters of which the first need not be a letter (e.g., 6603A is a legitimate unit name); the same name is placed in the location field of each RBR macro for this unit.

type — is a two-letter hardware type mnemonic as follows:

- AA 6603-I disk
- AB 6638 disk
- AC 6603-II disk
- AD 865 drum
- AP 854 disk pack drive
- AX Extended Core Storage

chan1 — is the (decimal) number of the primary data channel by which the unit can be accessed.

chan2 — is the (decimal) number of the alternate data channel if any, or zero or blank if none.

contr — is the equipment number of the control unit.

unit — is the unit number. For a 6603, unit is always zero. For a 6638, unit is 0 or 1. For a drum or disk drive, unit is 0 to 7.

sysres — is 1 if the system library is to be placed on this unit by the Dead Start package, or 0 or blank otherwise.

driver — is normally blank, but may be the three-character name of the PP program to be used as the Stack Processor for this device.

onoff — is 1 if the unit's EST entry is to be assembled as turned off (bit 23 set), and 0 or blank otherwise.

EQUIPMENT STATUS TABLE (EST)

	ASSIGNMENT		CHAN bb	CHAN aa	CHAN dd		CHAN cc		HARDWARE MNEMONIC	EQUIP #	0	UNIT #
3000 NON ALLOCATABLE	ASSIGNMENT		CHAN bb	CHAN aa	CHAN dd		CHAN cc		HARDWARE MNEMONIC	EQUIP #	0	UNIT #
6000 NON ALLOCATABLE	ASSIGNMENT		00	CHAN.	SYNC EQUIP #	0	UNIT #	ON OFF	HARDWARE MNEMONIC	0000		
3000 ALLOCATABLE	4	ZZZ	CHAN bb	CHAN aa	EQUIP #	0	UNIT #	ON OFF	HARDWARE MNEMONIC	DST ORDINAL		
6000 ALLOCATABLE	4	ZZZ			EQUIP #	0	UNIT #	ON OFF	HARDWARE MNEMONIC	DST ORDINAL		

242322

ASSIGNMENT - 0000 : NON ALLOCATABLE, AVAILABLE
000n : NON ALLOCATABLE, CONTROL POINT n
2000: NON ALLOCATABLE, UNAVAILABLE

ZZZ
- 000 = AVAILABLE / OFF AND NON ACTIVE — AVL/OFF
- xxn = ASSIGNED TO CONTROL POINT n — AOn
- 010 = ACTIVE FILES ON UNIT — BSY
- 020 = AVAILABLE FOR PRIVATE ALLOCATION — PVT
- 030 = UNASSIGNED, PRIVATE FILE ALLOCATED — PBS
- 040 = UNLOADED — UNL
- 110 = ASSIGNED TO SYSTEM AT DEAD START — SYS
- 210 = ASSIGNED TO PERMANENT FILE DIRECTORY — PFD
- 310 = ASSIGNED TO SYSTEM AND P.F. DIRECTORY — PFSY

HARDWARE MNEMONIC

CR	CARD READER	DS	DISPLAY CONSOLE	MX	MULTIPLEXOR
CP	CARD PUNCH	MT	½ INCH MAG. TAPE	TC	3266/3118 DSC
LP	LINE PRINTER-5H	LQ	LINE PRINTER-512	YC	6676 DSC
AA	6603 I DISK	SC	6673/6674 D.S.C.	DC	6671 DSC
AB	6638 DISK	AC	6603 II DISK	AD	865 DRUM
AX	ECS	AP	854 DISK DRIVE	IX	RESERVED FOR INSTALL
		AF	814 DISK		

d 6684/6681 on channel 0 = 6681 1 = 6684

2.6 Channel Status Table

The Channel Status Table contains one word per channel. If a channel is not reserved the format of the word is:

0000	CHANNEL NUMBER	unused	*	*

* = location of the word itself

If a channel is being used by a peripheral processor program, the format of the word will be :

0000	CHANNEL NUMBER	unused	*	PPOR

PPOR = PP Output Register Address

The channel number for a particular equipment is obtained by a peripheral processor program from the Equipment Status Table (EST) entry for that equipment. The program transmits its request for that channel to MTR (M.RCH). If the last byte of the entry corresponding to that channel shows that the channel is available, MTR enters in this byte the PP output register address and notifies the requestor that the channel has been assigned. When the requesting processor completes its operation on that channel it will drop the channel assignment (directly or via its resident) by resetting the last byte to tis primary value.

The length of the Channel Status Table is currently 24B. This includes the 12 hardware channels and the 4 following pseudo-channels:

Pseudo-channel 14_8 controlling the access to the File Status Table (FST)

Pseudo-channel 15_8 controlling the access to the File Name Table (FNT)

Pseudo-channel 17_8 controlling the access to the Record Block Table (RBT)

Pseudo-channel 23_8 representing the dump pseudo-channel.

Peripheral processor programs request these pseudo-channel assignments in the same manner as data channel are requested.

Note: Not all accesses to the FNT/FST entries require channel reservation: the function of the interlock scheme is to prevent two (or more) processors from attempting to modify the same entry at the same time. Pseudo-channel reservations are required in the following cases:

- whenever an entry is added to the FNT/FST
- whenever a file is assigned to a control point (FNT entry modified)
- whenever the buffer status byte is initialized at the beginning of an operation (FST entry modified)

2.7 Mass Storage I/O Processing

In order to comprehend the functions of the various tables involved, the SCOPE 3 mass storage I/O processing and the method of allocation must be understood.

Consider the 6603 disk; the 6603 contains 128 cylinders at each of 8 head group selections (1024 tracks). Each track is divided into 64 CM word sectors. In order to assign disk space to each of the several files existing at any one time within the system, a unit of allocation (called a record block) is selected. The SCOPE 3 standard unit of allocation on a 6603 disk is a halftrack; each physical track is considered to be composed of two half-tracks, one consisting of the odd-numbered sectors and the other consisting of the even-numbered sectors. Other units of allocation might be selected according to rules which are described elsewhere.

The first table of concern is called the Record Block Reservation table (RBR). An RBR contains a set of 2048 bits, each of which represents a single record block. If a bit is zero, the corresponding Record Block (RB) is unassigned and available for assignment; if a bit is one, the corresponding RB is unavailable for assignment (i.e., it is either already assigned to some file or physically unusable).

Each mass storage unit may have one or more associated RBR tables. The bits contained within an RBR represent a section of the disk which is to be allocated in a particular way. For instance, one disk may have two RBRs, the first of which contains bits for a part of the outer zone which is to be allocated by full track; the second RBR would contain bits for the remaining portions of the disk, which are to be allocated in half-tracks.

The bits in an RBR must have some definable relationship to each other, such that given a physical first word address, the stack processor can compute from the position of any bit in the RBR the physical address of the record block represented by the bit.

Understanding of the Record Block Table (RBT) is also required. Each mass storage file within the system has an associated RBT. All RBTs are kept in high central memory.

When a mass storage file is initiated, a single two-word RBT entry is assigned; additional entries will be assigned if needed. Each entry is divided into ten twelve-bit bytes, some of which are used as pointers to additional entries, other tables, etc.

The remaining bytes each contain the number of a particular record block assigned to the file, in the physical order of their assignment; i.e. the first byte contains the number of the first record block, and so forth.

A record block number in an RBT is the ordinal of the bit in an RBR which represents that record block.

The File Name Table (FNT) contains among other things, the seven-character display code file name, the device type on which the file resides, and file position information, expressed as pointers to the file's RBT.

All three of the above tables reside in the system portion of central memory and hence are protected. There is one other table which is involved in file processing. This table is called the File Environment Table (FET). The FET resides with the field length of the job using the file. The FET contains, among other things, the seven-character file name and the buffer pointers.

To illustrate the use of these tables, consider the creation of a file on disk. Assume there is only one standard RBR. A program issues a call to the system with the address of the FET. The FET code and status bits are set for a write function. An FNT entry is created, containing the allocation type from the FET. The RBR is searched for an available record block (bit value equals zero). The lowest-numbered available record block is selected, so that files tend to congregate at one end of the unit. When a record block is located, its RBR bit is set to one and the number of the bit is entered into a byte of the RBT. Writing begins. When the circular buffer is emptied, the position within the record block is noted in the FNT. Subsequent writes will continue from this position. When the record block is completely filled, the RBR will be searched for another available bit. This time, the RBR is searched end-around starting at the current position, to make logically consecutive record blocks within a file physically as close together as possible. Processing continues in this manner.

Positioning within a record block is kept in terms of physical record units (PRUs). On a 6603 disk, a PRU is a single sector. Thus, to specify completely the location of a record in a mass storage file, three numbers are needed: the RBT word-pair ordinal, the byte number of the byte within that word-pair which contains the record block number. and the PRU number within that record block. Now, consider a random file; i.e., a mass storage file to be written with an index. Within the FET, an index buffer address may be specified. The index buffer resides within the field length of the program. In this case, file writing is done as described above with one additonal step. Before the physical writing begins, the current file position as noted in the FNT is inserted into the index. Note that the file position is effectively a "logical" address in that it is a PRU number within the file rather than a physical address within the device. Hence no other file may be accidentally referenced. In addition, the index is device-independent; a random file can be copied from one mass storage device to another, or even to magnetic tape and back to mass storage, without special treatment of the index records even though mass storage device types differ in logical and physical addressing characteristics, because the PRU size is the same (64 CM words) for all mass storage device types.

File reading proceeds in a similar manner. Sequential files are read using the file position from the FNT. The position from which to begin reading of an indexed file is specified in the FET.

The contents of an index are essentially independent of the physical device; i.e. the record addresses have no set relationship to a record block or physical record unit, other than that each record begins in a new PRU. A single record block may contain part of one, or multiple records. These records are called logical records; their length is defined by the creator of the file. A file index contains one entry for each logical record.

2.7.1 Record Block Reservation Table (RBR)

Each 38 word RBR serves the dual purpose of defining record blocks in a specific allocatable area, and identifying those record blocks within that area as available or not available. A maximum of 2,048 record blocks may be defined by one RBR.

Four restrictions are placed on the area defined by an RBR:

1. Unless inequality is forced by the design of the device, all RB's described in a single RBR must be the same size. For example, an RBR defining all half-tracks on a 6603 disk must contain both 64-PRU record blocks and 50-PRU record blocks. This is valid because of 6603 logic.

2. The area described by an RBR must all be on one unit, and there exists at least one RBR for each mass storage unit.

3. Record blocks must be defined in simple relation to the physical characteristics of the device on which they are located. For example, on a 6603 disk drive, allocation by 31=sector groups is not allowable.

4. The area described by an RBR must consist of not more than 131,072 PRU's organized into not more than 2,048 record blocks. Bits for up to 2048 record blocks are contained in words 2-37. The association of a particular bit to a record block number is accomplished as indicated in the following RB bit table diagram (page 2-23): The RB's are mapped by groups of 64 (100B) into a 32 CM-word table plus four words containing the overflow bits which did not fit in this table (4 bits per word).

The addition of an RBR is accomplished by inserting into CMR a statement of the following form:

```
NAME    RBR    RBCT,START,FAULTS,LIST,ALLOC
```

NAME must agree with that used in the associated EDST statement for this mass storage unit.

RBCT is the number of available record blocks which may be indexed by this RBR. For standard allocations, the value of RBCT depends on the device type as follows:

RECORD BLOCK RESERVATION (RBR) HEADER

	59	53	47	41	35	23	17	11	5 0
W.RBRTPA W.RBRUNT	C.RBRTPA EQUIP. TYPE	0	C.RBRUNT DST ORDINAL	UNIT NUMBER	FIRST RB NUMBER	C.RBRA aa	bb	cc	dd
W.RBRLAV	TOTAL # RB's IN THIS RBR		EST ORDINAL		NUMBER OF USABLE RB's	C.RBRLAV NUMBER OF AVAILABLE RB's		CDC	

aa, bb, cc, dd - PERMISSABLE ALLOCATION STYLES

00 - SYSTEM DEFAULT (SAME AS 03)

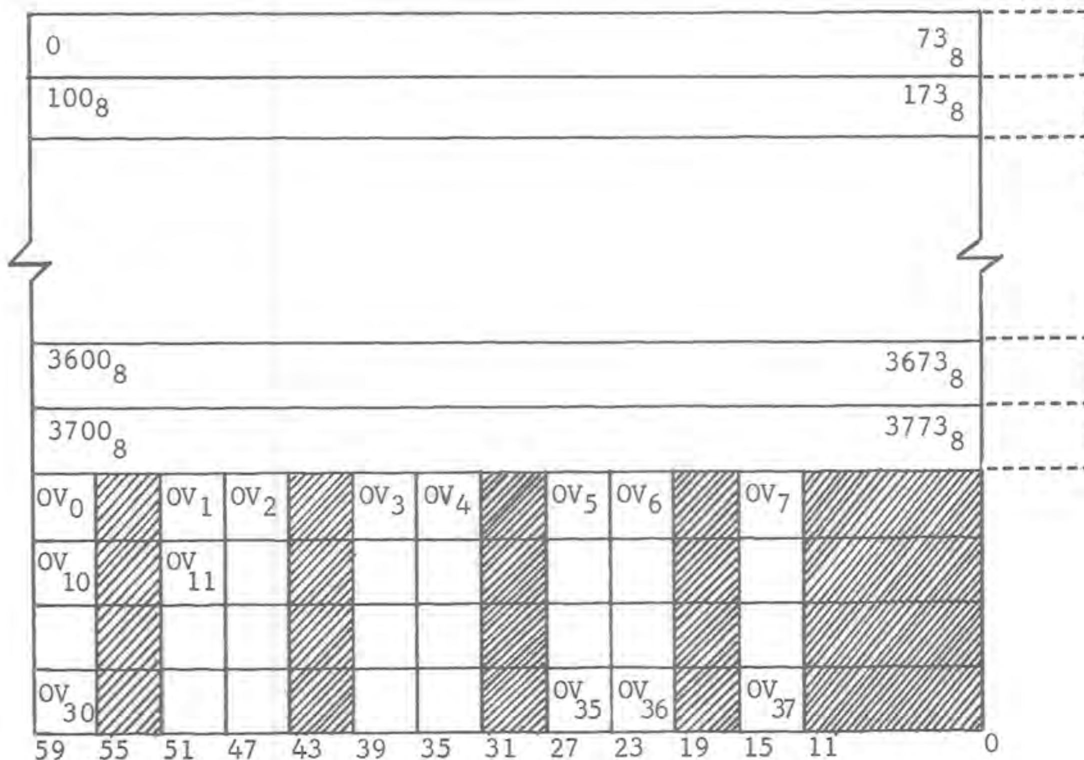
01 - FOR 6603 = INNER ZONE HALFTRACK (50 PRU/RB)
FOR 6638 = HALFTRACK (50 PRU/RB)
FOR 854 = PRIVATE PACK, TRACK (5 PRU/RB)

02 - FOR 6603 = OUTER ZONE HALFTRACK (64 PRU/RB)

03 - FOR 6603 = EITHER ZONE HALFTRACK (50 OR 64 PRU/RB)
FOR 6638 = SAME AS 01
FOR 863/865 = HALFTRACK (21 PRU/RB)
FOR 854: PRIVATE OR PUBLIC, TRACK (5 PRU/RB)

10 - FOR 6603 OR 6638 : RESPOND (8 PRU/RB)

RECORD BLOCK BIT TABLE



OV_i contains the 4 bits for RB's $100_8 i+74_8$ through $100_8 i+77_8$

type	RBCT (decimal)
AA	2048
AB	2048
AC	2048
AD	512
AP	2000
AX	defined at dead start time (allocatable block size divided by IP.ECLRB).

START is the physical start address of the RB group. It will be nonzero for nonstandard allocations to be available in later version.

FAULTS is the number of RBs indexed by this RBR which are not physically available.

LIST if nonblank indicates that the installation will preset the content of the RBR. In this case, only the two word header is generated by the RBR statement. If LIST is missing, 36 zero words are generated.

ALLOC defines the allocation type. Its form is

aabbccdd

where each term ii specifies a legimate allocation within the portion of the device indexed by this RBR.

01 50 PRU's/RB for 6603 and 6638 disk units; same as 03 for other mass storage device types

02 64 PRU's/RB, defined only for 6603 disk units

00 free allocation; when the user does not specify allocation in an OPEN call the file may be assigned to any RBR with some ii=00. In this case, allocation aa is assigned to the file after the RBR is chosen, thus "aa" should not be 00.

03 free allocation; this value is used to permit continuation of a file from one device to another.

10 8 PRU's/RB; this allocation style is used by RESPOND for many of the files which it manages. (See below for further details.)

04 - 07 and 11 - 77 remain undefined at this time.

"aa" determines the actual RBR type for physical allocation purposes.

ALLOC need not be supplied. Default values are

for TYPE = AA or AC in the same-named EDST macro:
ALLOC = 03020100

for TYPE = AB, AD, or AP in the same-named EDST macro:
ALLOC = 03010100

for TYPE = AX in the same-named EDST macro:
ALLOC = 03030300

Allocation code 10

This is a non-standard allocation type developed for use by RESPOND. It is defined only for 6603-I, 6603-II, and 6638 disk units. Eight sectors are included in each record block. Two sectors are unused at the end of each 50-sector half track to prevent the carry-over of a record block onto a second half track. The RBR and RBT byte format for allocation type 10 expects 8 record blocks from each half track. At RBR generation time, certain bits must therefore be set on to prohibit the referencing on non-existing record blocks 6 and 7 in each 50-sector half track. The count of these pre-set bits should be included in the "FAULTS" parameter of the RBR macro, and parameter "LIST" must be used to permit the pre-setting of bits.

RBR Header Format

The first two words of each RBR are used to define the area described by the RBR and to supply information required to utilize it. See figure for the table format. Details of some areas follows.

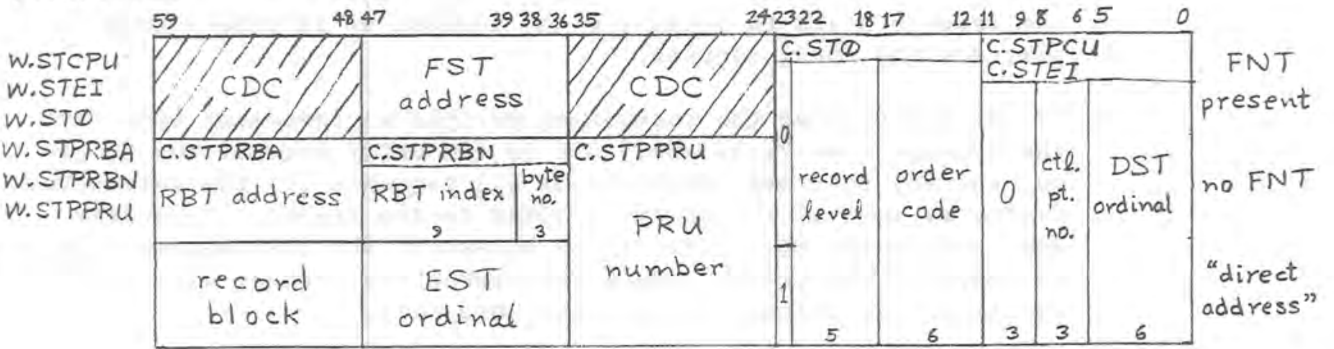
Word 0 (W.RBRTPA, W.RBRUNT): Byte 0 (C.RBRTPA) contains zeros in bits 0-5 and a device type code in bits 6-11. The defined device type codes are as follows (octal):

01	6603-I disk
02	6638 disk
04	6603-II disk
07	854 disk pack drive
12	865 drum
20	ECS

Byte 1 (C.RBRUNT) bits 6-11 contain the DST ordinal of the controller for this unit, and bits 0-5 contain the unit number. Byte 2 contains the value of the START parameter from the RBR macro; it is zero for standard allocations. Bytes 3 (C.RBRA) and 4 contain the allocation type codes admitted by this RBR. Word 1 (W.RBRLAV): Byte 0 contains the number of record blocks indexed by this RBR; i.e., the RBCT parameter in the RBR macro. Byte 1 contains the EST ordinal for the mass storage unit described by this RBR. Byte 2 contains the number of usable record blocks; i.e. the value of RBCT - FAULTS. Byte 3 (C.RBRLAV) contains the number of record blocks currently available for assignment; it is equal to byte 2 initially, and is decremented and incremented dynamically as record blocks from this RBR are assigned to files and released. The RBR is said to be full when this byte is zero. Byte 4 is reserved for future system use. No space in the RBR format is reserved for use by installations.

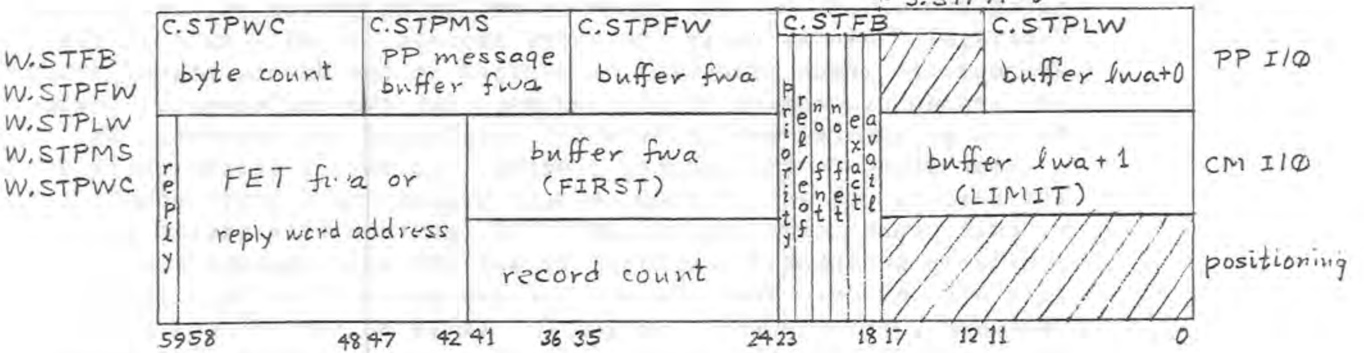
Request Stack Entry

First Word



S.STF = 6
S.STFPRI = 5
S.STFEØF = S.STFREL = 4
S.STFNTP = 3
S.STFETP = 2
S.STFXCT = 1
S.STFA = 0

Second Word



"direct address" defined only for PP I/Ø orders with no-fnt and no-fet flags set.

order code:

CM I/Ø orders

00	Ø.READ	READ into CM
01	Ø.RDSK	READSKP into CM
02	Ø.RCMPR	drop 1st 3 words
03	Ø.RDNS	READNS into CM
04	Ø.WRT	WRITE from CM
05	Ø.WRTR	WRITER/F from CM
06	—	
07	—	

PP I/Ø orders

10	Ø.RDP	READ into PP
11	Ø.RDPNP	drop 1st 15 bytes
14	Ø.WRP	WRITE from PP
15	Ø.WRPR	WRITER/F from PP

Positioning orders

12	Ø.SKF	SKIPF n logical records
13	Ø.SKB	SKIPB n logical records
16	Ø.BFRU	BKSPRU n PRU's
17	Ø.RCHN	Release chain (EVICT)

reply: 0 = normal FET, 1 = one-word FET (reply word) – must also set no-fet flag.
priority: 0 = normal, 1 = high priority
rel/eof: reading: 0 = normal, 1 = release RB's as read; writing: 0 = WRITER, 1 = WRITEF
no fnt: 0 = first word has first format, 1 = first word has second or third format
no fet: 0 = FET present, 1 = no FET (LIMIT = lwa+2 for writing, lwa+3 for reading)
exact: 0 = normal, 1 = don't look ahead if reading or don't skip to EØI before writing
avail: 0 = normal, 1 = requesting PP is available for use as stack processor

2.7.2 Request Stack

A. Request Generation. The peripheral processor program requesting an allocatable device I/O function must construct in the two high order CM words of its communication area a "stack entry", and in its output register a "stack entry request". Construction of these records involve a search of RBR's to locate the appropriate device for the entry and the entry request, and also a search of the request stack to determine the address of an unused word pair for the entry request.

If the function of the requesting peripheral processor ends with the request, the "available" bit in the entry request may be set on, thereby insuring execution of all requests for the referenced device as soon as the entry is added to the stack. If monitor does not accept the entry, it is necessary for the requesting program to research the request stack and repeat the request with the resultant new address of an empty word pair.

B. Stack Updating. Upon recognizing a stack entry request, monitor will check the word pair addressed in the request. If the word pair is not empty, it will refuse the entry. If the word pair is indeed empty, monitor will insert the entry into the word pair. It will then add one to the entry count field of the appropriate device status entry, and will test the "control PP" field of the device status entry to determine if the stack is being processed. If the stack is not being processed, and if the "available" bit is on in the entry request, it will call in the appropriate stack processor as defined in the device status entry and assign processing of all requests for the preferencial device to the peripheral processor which originated the request. At a later point in the monitor program, all device status entries are tested. If any of them require processing ("entry count" ≠ "exit count" and "control PP" = 0), monitor will assign an available peripheral processor to satisfy all requests for the relevant device. When the monitor assigns a PP to be a stack processor, it sets the "control PP" field of the DST entry non - zero, and directs the PP to load and execute program 1SP, with the DST ordinal in the input register. 1SP loads the driver overlay for the device type indicated by the DST entry, and commences to process all request stack entries for all mass storage units on the controller associated with the DST entry, and no others. 1SP remains in its PP until it can find no further requests in the stack directed to its DST entry, and the PP is needed for some other task, and at least one other DST entry has non - zero in its "control PP" field.

C. Stack Processing. The stack processor program selected will search the common request stack for the request most easily satisfied. Requests requiring no actual disk operations (backspace PRU, release RBT chain, and skip forward or backward with record count = 777777B) will be executed first. The remainder of the request will be selected on the basis of minimal head switching and repositioning. It is important to recognize at this point

that a stack processor does not establish a sequence of requests to be satisfied, it merely searches for the one request most easily satisfied. As a result, it is possible that a request involving re-positioning may wait for processing while several more easily satisfied requests for the same device are added to the stack and processed. The wait time will be minimized by the fact that an easily satisfied request requires little time and hence there is little opportunity for additions to the stack during its execution. In addition, a priority incrementing scheme ensures that a request with large access time will not be bypassed indefinitely; eventually its priority will become great enough to override access time considerations.

It is also possible for a stack processor to generate a stack entry and stack entry request. This can occur if it discovers that the logical disk record it is reading is continued on another device, or if it discovers that all record blocks in the area in which it is writing have been assigned, or if head repositioning is necessary to continue the current operation (so that requests involving the current position can be satisfied before moving the heads), or if more RBT space in CM must be allocated by monitor. Timing or other circumstances may cause a stack processor to terminate prior to completion of all requests for a device. This will not be a common occurence, and in all likelihood it will only result in a reassignment by monitor of another processor to the device.

Prior to execution of any request that references a central memory buffer or FET or reply word, the stack processor must attach itself to the control point which originated the request. If the storage move flag for that control point is set, the stack processor returns the request to the stack, detaches itself from the control point, and selects another request.

D. Stack Entry. The request stack area contains two types of entries, each of which occupy two words. The first N.DEVICE word pairs are called device status entries (described in 2.7.3 below). The remaining word pairs are stack entries: they are available for all pending requests for all allocatable devices within the system, irrespective of device type.

The Stack Entry is a two CM word request for a function involving a mass storage device. All stack entries are made by peripheral processor programs. Monitor accepts the entries and adds them to a common accessivly device request stack. Monitor then returns to the message buffer, an indication of whether or not the request was accepted; PP resident will reiterate its search until the request is entered. See figure for a diagram of the Request Stack entry. For the first word, there are three possible cases: 1. FNT is present; 2. no FNT present; and 3. direct address. These affect the contents of bits 24-59, as follows. In case 1, the presence of an FNT address, referencing the logical file to be processed, is signaled by a zero value in the FNT bit in the "Flag bits" field (bit 21 of the 2nd word of the entry). The

address of the middle word of the 3-word FNT entry appears in bits 36-47. Bits 24-35 are unused. Bit 48-59 contain zero values. Case 2 is indicated by a 1 in the FNT bit and a zero value in the direct bit (bit 23 of the first word). When this situation exists, bits 48-59 contain the RBT word pair number, bits 39-47 contain the number of the RBT word pair within the chain of RBT word pairs for this file, bits 36-38 contain the RB byte number within this RBT word pair, and bits 24-35 contain the PRU number within that Record Block. This information, normally found in the FNT, is carried here since no FNT exists for the reference file. Case 3 exists when the FNT, FET and direct bits all contain a 1 value. This occurs only when reading or writing labels on disk packs, i.e., there is no RBT chain from which the RB and EST numbers can be obtained. Bits 48-59 then contain the RB number, bits 36-47 contain the EST ordinal and as in case 2, bits 24-35 contain the PRU number.

Bits 0-23 of the first word are identical for all cases: As mentioned, bit 23 is the direct bit. It is 0 under normal circumstances and 1 for direct access when reading and writing multi - sortage labels. Bits 18-22 indicate the record level; bits 12-17 indicate an order code, which denotes the function desired.

Possible order codes:

00	O.READ	READ into CM
01	O.RDSK	READSKP into CM
02	O.RCMPR	READ into CM, drop first three CM words of record (for reading from library)
03	O.RDNS	READ NON STOP into CM
04	O.WRT	WRITE from CM
05	O.WRTR	WRITER/WRITEF from CM (R or F determined by "EOF" flag bits)
06		unused, reserved for CDC
07		unused, reserved for CDC
10	O.RDP	READ into PP core
11	O.RDPNP	READ into PP core, drop first three CM words (for overlays)
12	O.SKP	SKIPF
13	O.SKB	SKIPB
14	O.WRP	WRITE from PP core
15	O.WRPR	WRITER/WRITEF from PP core (R or F determined by EOF flag bit)
16	O.BPRU	BKSPRU
17	O.RCHN	EVICT

Bits 9-11 must contain zeros; bits 6-8 contain the number of the control point from which the request originated; bits 0-5 contain the Device Status Table ordinal.

The format of the second word of the request depends on the order code in the first word. The first format is used for a PP I/O request (order code 10, 11, 14, or 15). Byte 0 (C.STPWC) initially contains the number of 12-bit bytes to be written or the buffer size for reading, and is decremented as each PRU is transmitted. Byte 1

(C.STPMS) contains the first word address of the requesting PP's message buffer; this address+2 is the address of the "communication word" which is to PP I/O what the FET is to CM I/O. Byte 2 (C.STPFW) contains the first word address of the buffer area in the requesting PP's memory. Byte 4 (C.STPLW) contains the last word address (not lwa+1) of the buffer area in the requesting PP's memory; note that this is lwa+0 rather than lwa+1, and therefore a zero length logical record cannot be written from a PP.

The second format is used for a CM I/O request (order code 00 through 05). If bits 59 and 20 are both zero, bits 42-58 contain the address (relative to RA) of the first word of the FET; only the code/status field, IN and OUT pointers, and EP flag bit are referenced by Stack Processor. If bits 59 and 20 are both ones, bits 42-58 contain the address (relative to RA) of a "reply word" which is treated as the first word of an FET; i.e., effectively a one-word FET. If bit 59 is zero and bit 20 is one, bits 42-58 are ignored. Bits 24-41 contain the first word address (relative to RA) of the buffer area; this is normally equal to the FIRST pointer in the FET. Bits 0-17 contain the last word address+1 (relative to RA) of the buffer area; this is normally equal to the LIMIT pointer in the FET. When the no-FET flag (bit 20) is set, bits 0-17 contain the last word address+2 (relative to RA) of data to be written or the last word address+3 (relative to RA) of area into which data can be read.

The third format is used for a positioning request (order code 12, 13, 16, or 17). Bits 42-59 and 0-23 are as for CM I/O requests. Bits 24-41 contain the logical record count for order codes 12 and 13, the PRU count for order code 16, and is ignored for order code 17.

For all second word formats, bits 18-23 contain various flag bits. The system symbols S.xxx are shift counts for referencing these bits. Bit 23 (bit S.STF+S.STFPRI if byte C.STFB) is the request priority flag; setting this flag gives the request higher priority than all requests in which the priority bit is zero. Bit 22 (bit S.STF+S.STFREL or S.STF+S.STFEOF of byte C.STFB) is used in two ways. For read and skip-forward order codes (00-03 and 10-12) it causes releasing of each record block as soon as it has been read or skipped; when the file is continued to another device or end-of-information is reached, the entire RBT chain up to that point is released. For write end-of-record order codes (05 and 15) a zero in this flag bit means write end-of-record and a one means write end-of-file. Bit 21 (bit S.STF+S.STFNTP of byte C.STFB) is the no-FNT flag; zero means the first word of the request has the first format, one means the first word has the second or third format depending on the direct flag (bit 23 of the first word). Bit 20 (bit S.STF+S.STFETP of byte C.STFB) is the no-FET flag; it must be one for PP I/O requests, and for other requests it affects the interpretation of bits 42-59 and 0-17 of the second word as explained above. Bit 19 (bit S.STF+S.STFXCT of byte C.STFB) is the exact flag; for read to CM requests (order codes 00-03) with the no-FET flag (bit 20)

set, setting the exact flag bit causes Stack Processor to stop reading when the buffer is completely full (normally, it reads the next PRU because it might be a zero-length PRU which will always fit into the buffer); for write requests (order codes 04, 05, 14, or 15), Stack Processor skips to the file's end-of-information before beginning to write if the exact flag is zero, but setting the bit nonzero causes writing to start at the file's current position (i.e., a rewrite in place operation). Bit 18 (bit S.STF+S.STFA of byte C.STFB) is the available flag; setting this bit causes R.EREQS to jump to R.IDLE and MTR to release the PP when the function M.EREQS is accepted.

2.7.3 Device Status Table (DST)

The second type of entry in the Request Stack is the Device Status Table Entry, which, like the Request Stack Entry, consists of two words. These words contain information regarding the status of a specific I/O device. They are static and are usually defined at assembly time. The ordinal of this entry (beginning with one) is frequently referred to as the device number.

The first word is formatted as follows.

Bits 48-59 are reserved for CDC.
Bits 42-47 contain the driver name (see figure).
Bits 36-41 are for installation.
The Stack Entry Count is in the 12 bits 24-35. It is incremented by monitor each time an entry is added to the common request stack which refers to the device referenced by this device status entry. It is used in conjunction with the exit count to determine the need for further processing of related stack entries. Bits 18-23 and 12-17 provide the numbers of the alternate and primary channels of the device, respectively. Bits 0-11 contain the DST ordinal, used in matching RBRs to devices.

The second word contains in the 0th and 1st bytes the head positions of units 0 and 1, respectively. These bytes provide the last location of the respective read/write head on the I/O device to which this status entry refers and re used to determine which stack entry requires the least repositioning for processing. In the case of single access devices, byte 1 is not used and byte zero represents the position of the only read/write head. For devices with more than two units, Stack Processor uses function codes to determine the head positions of the remaining units.

The 2nd byte (bits 24-35) contains the exit or stack completion count. This 12 bit count is incremented once by 1SP each time a request for activity involving the related device or RBRs is satisfied.

In byte 3, bits 23-18 are reserved to installations, bits 17-15 contain the 6681 converter number (0 to 3) for a 3000-series device and zeros for a 6000-series device, and bits 14-12 contain the equipment number.

The 4th byte of second word of the DST is used for the PP assigned flag. It is set to non-zero by MTR when a PP is assigned to 1SP, and zeroed by 1SP when the PP is dropped.

DEVICE STATUS TABLES (DST) ENTRY

59–47	47–41	41–35	35–23	23–17	17–11	11–0	
////	DRIVER NAME	INST.	ENTRY COUNT	ALTERNATE CHANNEL	PRIMARY CHANNEL	DST ORDINAL	
HEAD 1 POSITION	HEAD 2 POSITION		EXIT COUNT	INST	[illegible] / 0 [illegible]	NON-ZERO IF A PP IS ASSIGNED	9000 6000

THERE IS ON DST ENTRY FOR EACH MASS STORAGE CONTROL UNIT.

DRIVER NAME - A SINGLE LETTER IN DISPLAY CODE, USED TO FORM A DRIVER OVERLAY NAME SSX. THE LETTERS ARE:

- P 6603-I DISK
- Q 6638 DISK
- R 863 OR 865 DRUM
- S 854 DISK PACK
- T 6603-II DISK

NOTE THAT EACH DST ENTRY POINTS TO ITSELF, IN BYTE 4 OF WORD 1. IF THE CONTROLLER CAN HAVE MORE THAN TWO UNITS (E.G. DRUMS AND DISK PACK), BYTES 0 AND 1 ARE THE HEAD POSITIONS OF UNITS 0 AND 1 RESPECTIVELY, AND THE CURRENT POSITIONS OF ANY REMAINING UNITS MUST BE OBTAINED VIA HARDWARE STATUS FUNCTIONS

2.7.4 Record Block Table

The Record Block Table area (RBT) is a collection of individual file chains, one for each file on an allocatable device currently recognized by the system. A maximum of 8192 CM words may be occupied by all the RBT's active at any one time. The RBT area expands and contracts by 100_8 - word blocks as files are created and released.

The RBT area starts in the highest-numbered word of central memory and expands downward. Each entry is two consecutive words. RBT entries are addressed by RBT word pair numbers. If the installation has M words of central memory, RBT word pair number N is absolute words M-2N and M-2N+1. When a word pair is released (e.g. by dropping or EVICTing its file), it is placed in an "empty chain" of currently-unused RBT word pairs.

When a file is initiated, a single two-word RBT entry is assigned to that file; additional entries are assigned as needed. Each entry is divided into 10 12-bit bytes, some of which are used as pointers to additional entries, other tables, etc. An RBT need not be continuous, the first byte of each CM word pair supplies the word pair number of the next entry in the file's table. The remaining bytes each contain the number of a particular record block assigned to the file in the physical order of their assignment; up to 8 RB's may be referenced by each 2-word entry.

The eight byte types which appear in a RBT entry are arranged as follows:

WORD 1	Bits		
C.RBTWPL	59-48	X byte	RBT link. The RBT word pair number of the next word pair in this file's RBT chain. A zero X byte indicates that the file's RBT chain ends with the current word pair.
C.RBTRBR	47-36	Y byte	contains two fields:
C.RBTFB	47-39		RBR link. A nine-bit field which contains the number of the RBR for the logical file. This field is always present in the Y-byte of each RBT word pair. It may also be found in the high-order 9 bits of any of the bytes 0-7, in the event that the logical fiel is continued in an allocatable area defined by another RBR. In this case, the byte has zeros in bits 2 and 0 and a one in bit 1, to distinguish it from an RB-number byte.

RECORD BLOCK TABLE

FIRST WORD PAIR

S.RBTRBR ←3→ ; ←7→ S.RBTREL ; ←6→ S.RBTRND

C.RBTWPL NEXT RBT WORD PAIR	C.RBTRBR C.RBTFB RBR ORDINAL	FIRST RB BYTE	C.RBT... 0		ALLOC. TYPE	C.RBTPRU LAST +1 PR NUMBER	C.RBTLRB CUR.RB(RANDOM) RB2
C.RBTLPR CUR.PR(RANDOM) RB3	RB4		RB5			RB6	RB7

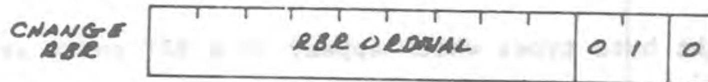
OTHER WORD PAIRS

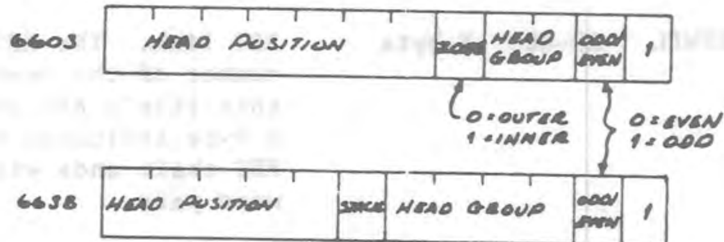
S.RBTRBR ←3→

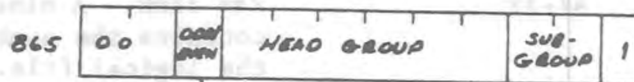
C.RBTWPL NEXT RBT WORD PAIR	C.RBTRBR C.RBTFB RBR ORDINAL	0	RB0	RB1	RB2
RB3	RB4		RB5	RB6	RB7

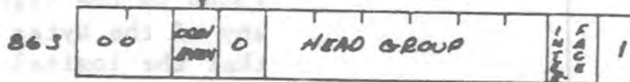
ALLOC. TYPE 01 = 50 PR'S/RB (6603 INNER ZONE)
02 = 64 PR'S/RB (6603 OUTER ZONE) SEE RBR
10_8 = 8 PR'S/RB (RESPOND)

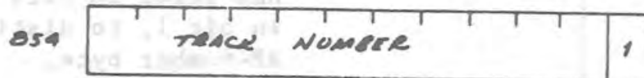
RB BYTES











TO GET PHYSICAL ADDRESS, DIVIDE BY 10 – QUOTIENT IS CYLINDER AND REMAINDER IS TRACK

Field	Bits	Byte	Description
S.RBTRBR	38-36		First RBT byte. A three-bit field containing the number of the first byte in the current word pair which contains an RB link. This will normally contain a zero (for byte 0) for all except the first word pair of an RBT chain. In the first word pair, the field will contain 2, if the file is sequentially accessed, or 4, if the file is randomly accessed. This field does not occur in bytes 0-7.
	35-24	0 byte	Flags and allocation type. This field is present only in the first word pair.
	35-32		are zero.
	31		Bit 31 contains the release flag, if set, record blocks are to be released after reading or skipping.
	30		Bit 30 is set if file is random.
	29-24		Bits 29-24 contain the allocation type as defined in the ERS.
C.RBTPRU	23-12	1 byte	Next PRU. The ordinal of the next PRU in the last record block of the file in which writing is to occur. This field may differ from the same field in the FNT, which references the PRU following the last on read or written. This field occurs only in the first word pair for a logical file
C.RBTLRB	11-00	2 byte	Previous last RB. For a randomly accessed file, the last RB assigned in the previous generation. This field occurs only in the first word pair of a randomly accessed file.
C.RBTLPR	59-48	3 byte	Previous next PRU. For a randomly accessed file, the last PRU assigned, plus one, in the previous generation. This field occurs only in the first word pair of a randomly accessed file.
	47-36 35-24 23-12 11-00	4 byte 5 byte 6 byte 7 byte	RB Links. The ordinals of bit positions in the RBR referenced by the last RBR link. (See word 1, bits 47-39.) In all word pairs other than the first in a chain, RB links will also be found in bytes 0, 1, 2 and 3. The high order eleven bits contain the ordinal; the low order bit is always on to distinguish an RB link from an RBR link which may also occur in bytes 0-7.

2.8 FNT/FST

FILE NAME AND FILE STATUS TABLES (FNT/FST)

FNT/FST GENERAL

The File Name Table (FNT) and the File Status Table (FST) comprise a single central memory table (FNT/FST) consisting of three central memory words. One such entry exists for each file which is to be referenced by a job at a control point. The first of the three words in each table is called the File Name Table (FNT), and the second and third words are the File Status Table (FST). The discussion to follow, concerns a single CM table (FNT/FST).

The FNT/FST is set up at the time that a file is created. In most instances, the overlay 2BP (see Buffer Parameter) builds the FNT/FST (if one has not already been built) in the course of its function of checking buffer parameters in a File Environment Table (FET). (See SCOPE Reference Manual Ch. 3)

The FNT/FST which provides linkage between user programs and all I/O and functions, is a system table not accessable by the user. Certain entries are found in both the FET and FNT/FST, in order that they be available to the user, and at the same time that the system be fully protected, e.g. the code and status word.

FNT/FST ORGANIZATION

The first (FNT) word:

Bits	Item
0-11	Priority. A 12-bit field defining the priority of the file.
12-14	Control point number associated with the control point to which this file is currently assigend.
15-16	File type code 00 INPUT 01 OUTPUT 10 Common 11 Local
17	Lock bit 0 File is available. 1 File is not currently available.
18-59	Logical file name, up to seven alphanumeric characters, left-adjusted.

FNT/FST. The second (first FST) Word:

The second word of the FNT/FST (that is, the first word of the FST) takes several forms dependent upon the nature and use of the file. The first and third words have the same form in all cases. Details are given in subsequent sections and the following figure.

FILE NAME TABLE

W.FTYPE (bit positions 59 54 53 48 47 36 35 24 23 17 11)

C.FNAME FILE NAME	C.FLPNUM C.FLOCK C.FTYPE	TYPE	CTL PT	C.FPRI · PRIORITY · RPF POINTER

S.FNTLK, S.FNTTYP

CARD READ FILE (42 41, 17)

C.FEQP 60	RECORD COUNT	CURRENT RECORD	EST ØRD	C.FLBL	CKPT	CARD COUNT
	FET ADDRESS		C.FDC DISPOSITION CODE 12	C.FSC/C.FCS	SEC CODE 2	LAST CODE/ STATUS

CARD PUNCH FILE

C.FEQP 70		EST ØRD	C.FLBL 2 PC FLAG 6	CARD COUNT
	FET ADDRESS	C.FDC DISPOSITION CODE	C.FSC/C.FCS SEC CODE 2	LAST CODE/STATUS

INPUT FILE — NOTE A

C.FEQP EQUIP. CODE	r		C.FFRBA FIRST RBT WORD PAIR	ECS FL/1000$_8$		CM FL/100$_8$
C.FABT C.FTL	abort	flag	TIME LIMIT	C.FDC 0000		

MASS STORAGE FILE — NOTE B

C.FEQP EQUIP. CODE 6		C.FFRBA FIRST RBT WORD PAIR	C.FLRBWP CURRENT RBT WORD PAIR	C.FLRBRB CURRENT RBT ORD	CUR RBT BYTE	C.FLPRU CURRENT PR
	FET ADDRESS		C.FDC DISPOSITION CODE	C.FSC/C.FCS SEC CODE 2		LAST CODE/STATUS

TAPE

C.FEQP EQUIP CODE	x	l	d	C.FSDEV SECONDARY UNIT EST ORD.	C.FPDEV PRIMARY UNIT EST. ORD	C.FLBL CURRENT PR COUNT
	FET ADDRESS				C.FDC DISPOSITION CODE	C.FSC/C.FCS SEC CODE 2 LAST CODE/STATUS

MULTI-FILE TAPE — NOTE C

C.FEQP EQUIP CODE		LABEL	d 2	C.FSDEV SECONDARY UNIT EST. ORD	C.FPDEV EST ORDINAL PRIMARY UNIT			CURRENT FILE POSITION (DECIMAL DISPLAY CODE)
			CURRENT FILE ...	CURRENT FNT POINTER	C.FDC 0002	C.FSC/C.FCS	3 2	

LOCK. 0 = UNLOCKED; 1 = LOCKED   TYPE: 0 = INPUT; 1 = OUTPUT; 2 = COMMON; 3 = LOCAL
SEC CODE: 0 = OPEN ALTER; 1 = OPEN READ; 2 = OPEN WRITE; 3 = CLOSED

CK: CHECKPOINT
d: 0 = 556 bpi; 1 = 200 bpi; 2 = 800 bpi
x: 00 = NORMAL; 01 = SCOPE 20 FORMAT 11 = LONG RECORD 10 = S TAPE FORMAT
NOTES A: FILENAME IN W.FTYPE = JOB NAME
B: (C.FLRBWP) = 0 → C.FLPRU = SPECIAL ALLOC. 12 BIT ALLOCATION TYPE
C: FILE NAME IN W.FTYPE = MULTI FILE NAME

r: return
[illegible] 10 = 3400 TYPE

A. Case of Sequential Files

Two situaltions can be distinguished in the case of sequential files. First, each sequential file has an FNT/FST. Second, a sequence of files may be designated by a "supername" and be provided with an FNT/FST having special properties. The latter usage is of importance in the case of multi-file tapes.

1. Individual Sequentail Files - See Figure

2. Multi-File FNT/FST
In this case, the name in word one of the FNT/FST is the "supername" by which the sequence of files is known.

Note: The secondary device is specified for use in a "ping pong" type of operation in multi-tape files or "super-files".

B. Case of Random Files (Allocatable Devices)

See Figure

C. Case of Job files prior to their Assignment to a Control Point

The second FNT word is used to hold certain parameters from the job card of an input file, prior to its assignment to a control point. In all other respects it has the appearance of an entry for a random file. These parameters are removed by 1BJ when the job is assigned to a control point. 1BJ further alters this entry so that is corresponds to a rewound random file. The usage in this situation is shown below.

D. Special Usage Internal to 2RC and 2PC

See Figure

2.9 Installation Area

Reserved for installation use.

2.10 Dayfile Buffer Area

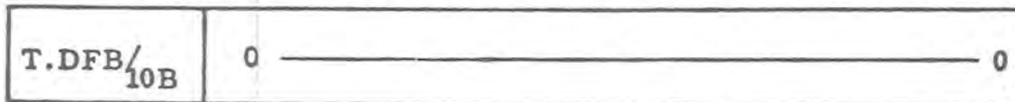
The Dayfile Buffer area contains 100B CM words. The first represents the System Dayfile File Environment Table followed by seven more FET, one each for the seven control points. The eight remaining words are buffers in a similar arrangement.

2.10.1 C. E. Error File Buffer

The C. E. Error File Buffer area contains 100B CM words. Error File entries are made via Monitor function M.DFM or from a CPU program via a call to PP routine MSG. This buffer is maintained in the same manner as the dayfile buffers.

DAYFILE BUFFERS

DAYFILE POINTER



DFB

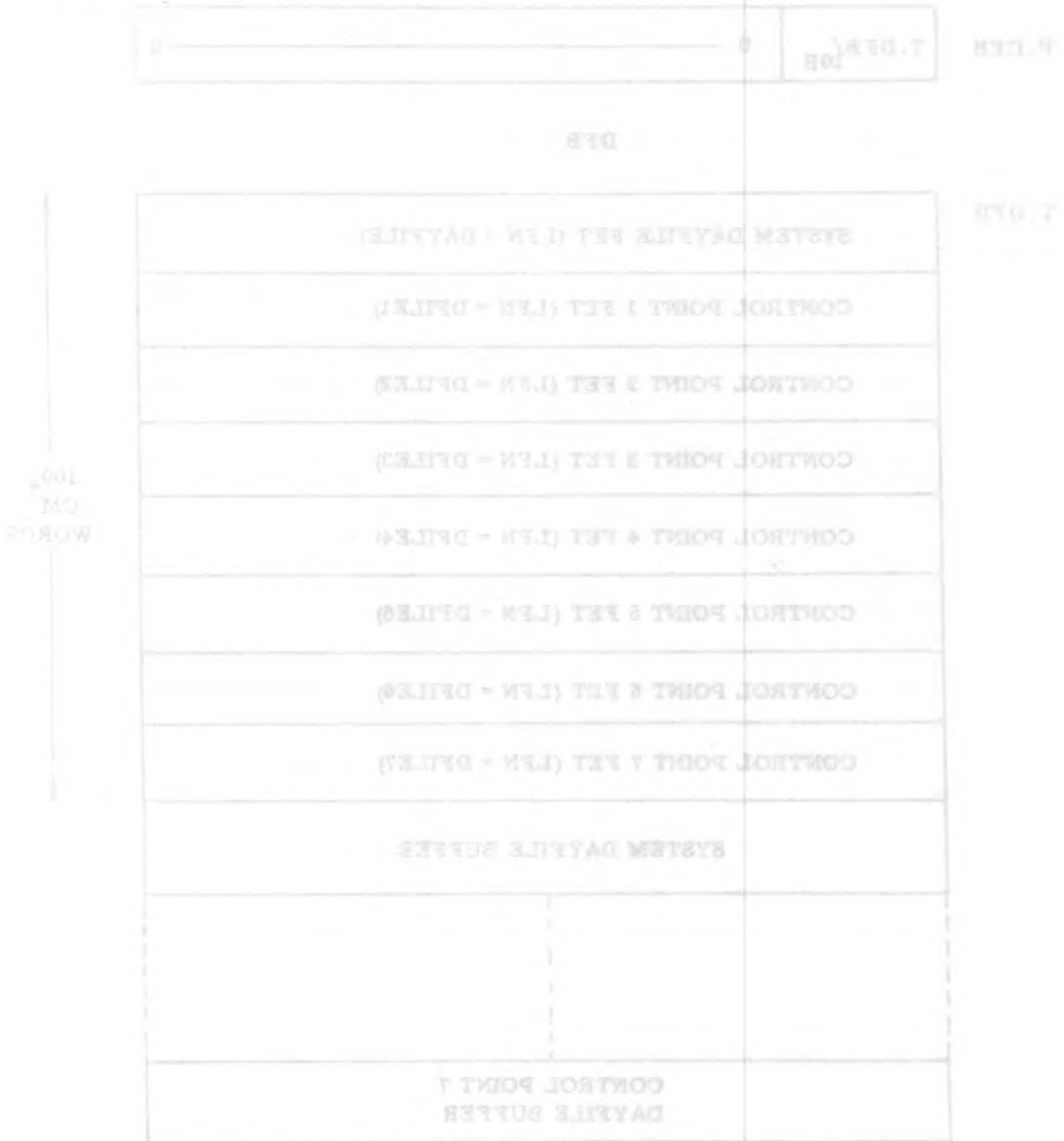
T.DFB

SYSTEM DAYFILE FET (LFN = DAYFILE)
CONTROL POINT 1 FET (LFN = DFILE1)
CONTROL POINT 2 FET (LFN = DFILE2)
CONTROL POINT 3 FET (LFN = DFILE3)
CONTROL POINT 4 FET (LFN = DFILE4)
CONTROL POINT 5 FET (LFN = DFILE5)
CONTROL POINT 6 FET (LFN = DFILE6)
CONTROL POINT 7 FET (LFN = DFILE7)
SYSTEM DAYFILE BUFFER
CONTROL POINT 7 DAYFILE BUFFER

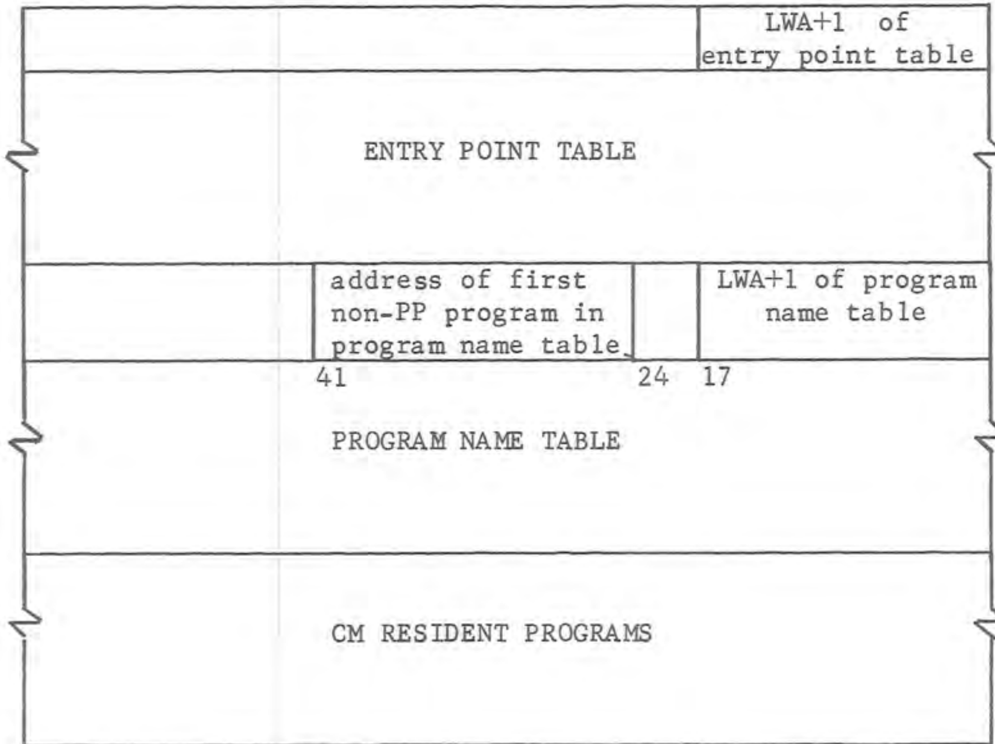
100_8 CM WORDS

2.11 Library Directory

The Library Directory is composed of three sections: The entry point table, containing 1-word entries; the program name table, containing 2-word entries; the program name table, containing 2-word entries; and the bodies of the CM resident programs.



LIBRARY DIRECTORY



ENTRY POINT TABLE

ENTRY POINT NAME	PROGRAM NUMBER
.	
.	
.	
.	
.	
59	17 … 0

PROGRAM NAME TABLE

CM RESIDENCE

59 … 18	17 … 0
PROGRAM NAME	LENGTH NOT INCLUDING 3 WORD HEADER

C.DIRPTR 0	TYPE	P	EDITION	C.DIRCMA 0	FWA IN CM	C.DIRRBN RBT ORD.	RBT BYTE	C.DIRPRU PR NUMBER
59	55	51	50	44	41 … 24	23 … 15	14 … 12	11 … 0

S.DIRPT
S.DIRPR

DISK RESIDENCE

59 … 18	17 … 0
PROGRAM NAME	LENGTH

C.DIRPTR 1	TYPE	P	EDITION	C.DIRUNT 0	DST ORD.	C.DIRRBA RBT ADDRESS	C.DIRRBN RBT ORD.	RBT BYTE	C.DIRPRU PR NUMBER
59	55	51	50	44	41	35	23	14	11 … 0

ECS RESIDENCE

59 … 18	17 … 0
PROGRAM NAME	LENGTH

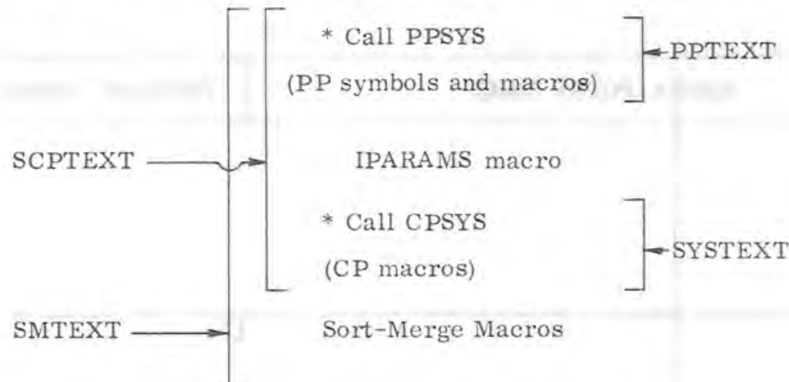
C.DIRPTR 2	TYPE	P	EDITION	FWA IN ECS	C.DIRRBN RBT ORD.	RBT BYTE	C.DIRPRU PR NUMBER
59	55	51	50	44	23	14	11 … 0

TYPE: 0 = PP 1 = CP RELOCATABLE 2 = CP OVERLAY 17_8 = ERT

P: 0 → FROM SYSTEM 1 → FROM SSSSSSU

2.12 SYSTEM DEFINITIONS

The following chart shows the system text organization:



The symbol definitions have one of the following forms:

```
Name   EQU   absolute expression
Name    =    absolute expression
```

where the = is the shorthand equivalent of the EQU instruction and the absolute expression is either an absolute value or an expression in which all the symbols have previously been assigned an absolute value.

PPTEXT definitions have the format:

```
i.mn
```

i Identifier; one or two characters identifying the category to which the symbol belongs.

mn Mnemonic; one to six characters suggesting the meaning of the symbol.

PPTEXT IDENTIFIERS

The following list describes the identifiers which appear in PPTEXT definitions:

Identifier	Description
C	Byte number within a CM word (0-4). C-identifiers are used for flags and parameters of 12 bits or less.
CP	Word number within the central processor resident area.
D	Direct cells (Section 1)
L	Lengths
LE	Length of table entries
M	PPU request of monitor (Section 7)
F	Error flag values
CH	Pseudo-channel assignments
N	Numbers (quantities of things)
O	Stack processor orders

Identifier	Description
OV	PPU Overlays. The mnemonic is the three character overlay name.
P	CM location of pointer words (Section 1)
R	PPU resident entry points (Section 1)
S	The number of bits a parameter must be right shifted to become right justified in a PPU word.
T	The first word address of CM tables. The system programmer should use the P. definition rather than access the table directly with the T. definition (Section 1).
W	Relative positions within CM tables (Section 1).

CH.CPA (20)
PSEUDO-CHANNEL FOR CONTROL POINT AREA INTERLOCK.

CH.DMP (23)
PSEUDO-CHANNEL USED BY DMP.

CH.FNT (15)
FILE NAME TABLE PSEUDO-CHANNEL;POSSESSION OF THIS CHANNEL INTERLOCKS WORD ONE OF FNT.

CH.FST (14)
FILE STATUS TABLE PSEUDO-CHANNEL;POSSESSION OF THIS CHANNEL PROVIDES AN INTERLOCK OF WORDS TWO AND THREE OF THE FNT (ALTERNATIVELY CALLED FST).

CH.INS (22)
PSEUDO-CHANNEL RESERVED FOR INSTALLATION USE.

CH.LIB (16)
LIBRARY DIRECTORY PSEUDO-CHANNEL; ONLY USED BY GP250 SOFTWARE PACKAGE (NOVA).

CH.PFM (21)
PSEUDO-CHANNEL USED BY PERMANENT FILE MANAGER.

CH.RBT (17)
RECORD BLOCK TABLE PSEUDO-CHANNEL.

C.APF (3)
BYTE OF WORD P.PFM1 CONTAINING THE FWA OF THE APF TABLE.

C.APFL (1)
BYTE OF WORD P.PFM1 CONTAINING THE LENGTH OF THE APF TABLE.

C.CKP (4)
BYTE OF WORD W.CKP CONTAINING THE NUMBER OF CHECKPOINTS TAKEN FOR THE JOB.

C.CPDFMC (0)
BYTE OF WORD W.CPDFMC CONTAINING DAYFILE MESSAGE COUNT FOR THE JOB WHEN COUNT IS ZERO, JOB IS ABORTED.

C.CPECFL (4)
BYTE IN WORD IN CONTROL POINT AREA CONTAINING NUMBER/1000B OF ECS WORDS ASSIGNED TO CONTROL POINT.

C.CPECSI (3)
BYTE OF WORD W.CPJCP CONTAINING ECS FIELD LENGTH FROM THE JOB CARD. USED IF JOB IS RERUN.

C.CPEF (1)
BYTE IN WORD IN CONTROL POINT AREA CONTAINING ERROR FLAG. IF ZERO, THERE IS NO ERROR AT THIS CONTROL POINT;OTHERWISE, C.CPEF MAY CONTAIN A VALUE DEFINED BY THE F.X SYMBOLS.

C.CPFL (4)
BYTE IN WORD IN CONTROL POINT AREA WHICH CONTAINS CENTRAL MEMORY FIELD LENGTH/100B ASSIGNED TO THIS CONTROL POINT.

C.CPFLAG (1)
BYTE WITHIN WORD W.CPFLAG IN CONTROL POINT AREA WHICH CONTAINS ABORT AND CLEAR FLAGS.

C.CPFLI (4)
BYTE OF WORD W.CPJCP CONTAINING CM FIELD LENGTH FROM THE JOB CARD. USED IF JOB IS RERUN.

C.CPFP (3)
BYTE OF WORD W.CPERT CONTAINING FLAGS FOR USE BY JOB PROCESSING. THE RESPOND AND EXPORT/IMPORT BITS ARE IN THIS BYTE.

C.CPFST (2)
BYTE IN WORD W.CPERT IN CONTROL POINT AREA. CONTAINS THE POINTER TO THE FST ENTRY OF THE INPUT FILE ASSIGNED TO THE CONTROL POINT.

C.CPLAI (2)
BYTE OF WORD W.CPLDR CONTAINING THE LOADER-ALREADY-IN FLAG.

C.CPLDR (3)
THE BYTE IN WORD W.CPLDR OF THE CONTROL POINT AREA WHICH CONTAINS FLAGS TO INDICATE THE VALUE OF THE MAP OPTION, WHETHER TO USE THE OLD OR NEW LOADER, AND FLAGS FOR DEBUG.

C.CPLIBP (4)
BYTE IN WORD W.CPCC CONTAINING A FLAG SET TO 1 WHEN 1AJ LOADS AN ABSOLUTE OVERLAY FROM THE SYSTEM LIBRARY.

C.CPNCSP (4)
BYTE IN WORD W.CPJNAM OF CONTROL POINT AREA. CONTAINS POINTER TO NEXT CONTROL STATEMENT TO BE PROCESSED.

C.CPNUM (1)
BYTE IN PP INPUT REGISTER WORD WHICH CONTAINS CONTROL POINT NUMBER ORIGINATING REQUEST.

C.CPOAE (4)
BYTE OF WORD W.CPOAE CONTAINING EST ORDINAL OF EQUIPMENT ASSIGNED BY OPERATOR.

C.CPOUT (3)
BYTE IN WORD W.CPOUT WHICH CONTAINS FLAG INDICATING JANUS CONTROL POINT.

C.CPPRI (0)
BYTE IN WORD W.CPPRI IN CONTROL POINT AREA. THIS BYTE DENOTES PRIORITY OF JOB AT THIS CONTROL POINT.

C.CPRA (3)
BYTE IN WORD W.CPSTAT IN CONTROL POINT AREA WHICH CONTAINS REFERENCE ADDRESS (RA)/100B OF CONTROL POINT.

C.CPREQ (0)
BYTE OF WORD W.CPOAE USED AS A FLAG TO DSD TO CALL REQ WHEN THE OPERATOR ASSIGNS EQUIPMENT.

C.CPRO (4)
BYTE IN WORD W.CPRO CONTAINING THE ROLLED OUT FLAG FOR THE JOB. 0 INDICATES NOT ROLLED OUT. 1 INDICATES ROLL-OUT INITIATED. 2 INDICATES ROLLED OUT. 3 INDICATES ROLL-IN INITIATED.

C.CPRPRI (2)
BYTE IN WORD W.CPOAE IN CONTROL POINT AREA WHICH CONTAINS RERUN PRIORITY.

C.CPSM (2)
BYTE IN WORD W.CPSM IN CONTROL POINT AREA WHICH CONTAINS STORAGE MOVE FLAG. MTR SETS THIS FIELD NONZERO WHEN STORAGE IS TO BE MOVED. ALL PP≠S AT THIS CONTROL POINT SHOULD PAUSE IF C.CPSM IS NONZERO.

C.CPSTAT (0)
BYTE IN A WORD W.CPSTAT IN CONTROL POINT AREA WHICH MTR USES TO NOTE STATUS OF CONTROL POINT

C.CPTIML (1)
BYTE IN WORD W.CPTIML IN CONTROL POINT AREA WHICH CONTAINS JOB TIME LIMIT.

C.CPTLI (1)
FIRST BYTE OF A TWO-BYTE FIELD IN WORD W.CPJCP CONTAINING THE TIME LIMIT FROM THE JOB CARD USED IF JOB IS RERUN.

C.CST (2)
BYTE IN WORD P.CST IN CMR WHICH CONTAINS FWA OF THE CHANNEL STATUS TABLE.

C.CSTL (3)
BYTE IN WORD P.CSTL IN CMR WHICH CONTAINS LWA+1 OF THE CHANNEL STATUS TABLE.

C.DIRCMA (1)
BYTE IN EACH WORD OF LIBRARY DIRECTORY WHICH CONTAINS CENTRAL MEMORY ADDRESS OF THAT LIBRARY ROUTINE.

C.DIRFWA (0)
LEFTMOST OF TWO BYTES IN P.LIB WHICH CONTAINS FIRST WORD ADDRESS OF LIBRARY DIRECTORY.

C.DIRPRU (4)
BYTE IN SECOND WORD OF A DIRECTORY ENTRY WHICH CONTAINS NUMBER OF FIRST PRU ASSIGNED TO RECORD.

C.DIRPTR (0)
BYTE WITHIN EACH WORD OF LIBRARY DIRECTORY WHICH CONTAINS PROGRAM TYPE AND RESIDENCE.

C.DIRRBA (2)
BYTE IN SECOND WORD OF A DIRECTORY ENTRY CONTAINING LINKAGE TO RBT WORD PAIR DEFINING RECORD IN WHICH THE RECORD STARTS.

C.DIRRBN (3)
BYTE IN SECOND WORD OF A DIRECTORY ENTRY CONTAINING ORIGINAL RBT WORD PAIR AND BYTE DEFINING BLOCK IN WHICH RECORD STARTS.

C.DIRUNT (1)
BYTE IN SECOND WORD OF DIRECTORY ENTRY CONTAINING PHYSICAL UNIT NUMBER (DST ORDINAL).

C.ECFLAW (2)
BYTES IN CMR WHICH CONTAIN ADDRESS AND LENGTH OF THE ECS FLAW TABLE.

C.ECST (4)
BYTES IN CMR WHICH CONTAIN ADDRESS AND LENGTH OF THE ECS INFORMATION TABLE.

C.ECSTAT (1)
BYTES IN CMR WHICH CONTAIN ADDRESS AND LENGTH OF ECS STATISTICS TABLE.

C.FABT (0)
BYTE OF FNT WORD 3 CONTAINING THE ABORT FLAG.

C.FADEV (3)
BYTE OF FNT WORD 2 CONTAINING FOR AN ALLOCATABLE DEVICE FILE TO WHICH NO RBT HAS YET BEEN ASSIGNED THE EST ORDINAL OF THE DEVICE TO WHICH IT HAS BEEN ASSIGNED.

C.FAPF (4)
BYTE OF FNT WORD W.FTYPE CONTAINING THE APF POINTER FOR A PERMANENT FILE.

C.FALLOC (4)
BYTE IN FNT WORD 2 CONTAINING ALLOCATION STYLE FOR A FILE NOT YET ASSIGNED TO MASS STORAGE.

C.FCB (2)
BYTE IN WORD IN FNT WHICH CONTAINS THE COMMON FILE CHANGE BIT.

C.FCIB (0)
LEFTMOST OF TWO BYTES IN WORD 3 OF THE FILE NAME TABLE PLANNED FOR FUTURE SYSTEM USE.

C.FCPNUM (3)
BYTE IN WORD 1 OF A FILE NAME TABLE ENTRY CONTAINING FILE≠S CONTROL POINT ASSIGNMENT.

C.FCS (3)
LEFTMOST OF TWO BYTES IN WORD 3 OF FILE NAME TABLE ENTRY CONTAINING CODE AND STATUS FIELD.

C.FDC (2)
BYTE IN WORD 3 OF FILE NAME TABLE ENTRY CONTAINING FILE DISPOSITION CODE.

C.FECFL (2)
BYTE OF FNT WORD 2 CONTAINING THE ECS FIELD LENGTH FOR A JOB IN THE INPUT QUEUE.

C.FEQP (0)
BYTE IN WORD 2 OF FILE NAME TABLE ENTRY CONTAINING FILE EQUIPMENT CODE.

C.FFL (4)
BYTE OF FNT WORD 2 CONTAINING THE CM FIELD LENGTH FOR A JOB IN THE INPUT QUEUE.

C.FFRBA (1)
BYTE IN WORD 2 OF FILE NAME TABLE ENTRY CONTAINING ADDRESS OF FIRST RBT WORD PAIR FOR A FILE ON ALLOCATABLE DEVICE.

C.FLBL (3)
BYTE IN WORD 2 OF FNT ENTRY. FOR MAGNETIC TAPE, BYTE C.FLBL CONTAINS UPPER 12 BITS OF CURRENT PHYSICAL RECORD UNIT (PRU) COUNT; LOWER 12 BITS OF PRU COUNT ARE IN BYTE C.FLBL+1. FOR PUNCHED CARDS, BYTE C.FLBL IS 12-BIT BYTE CONTAINING UPPER 5 BITS OF CARD COUNT WITHIN A RECORD BEING PUNCHED; LOWER 12 BITS OF CARD COUNT ARE IN BYTE C.FLBL+1.

C.FLOCK (3)
BYTE IN WORD 1 OF FILE NAME TABLE ENTRY CONTAINING THE LOCK BIT.

C.FLPRU (4)
BYTE IN WORD 2 OF FILE NAME TABLE ENTRY CONTAINING CURRENT PHYSICAL RECORD UNIT (PRU) POSITION OF A FILE ON AN ALLOCATABLE DEVICE.

C.FLRBEB (3)
BYTE IN WORD 2 OF FILE NAME TABLE ENTRY CONTAINING RBT ENTRY AND BYTE AT WHICH A FILE ON ALLOCATABLE DEVICE IS CURRENTLY POSITIONED.

C.FLRBWP (2)
BYTE IN WORD 2 OF FILE NAME TABLE ENTRY CONTAINING ADDRESS OF CURRENT RBT WORD PAIR FOR FILE ON ALLOCATABLE DEVICE.

C.FNAME (0)
LEFTMOST OF FOUR BYTES IN WORD 1 OF FNT ENTRY CONTAINING SEVEN-CHARACTER FILE NAME.

C.FPDEV (2)
BYTE IN FNT WORD 2 OF TAPE FNT ENTRY CONTAINING PRIMARY DEVICE NUMBER (EST ORDINAL).

C.FPRI (4)
BYTE IN FNT WORD 1 CONTAINING PRIORITY OF INPUT OR OUTPUT FILE.

C.FRRN (0)
BYTE OF FNT WORD 2 CONTAINING THE RERUN FLAG FOR A JOB IN THE INPUT QUEUE.

C.FSC (3)
BYTE IN FNT WORD 3 CONTAINING FILE SECURITY CODE, INDICATES WHETHER OR NOT FILE IS OPEN.

C.FSDEV (1)
BYTE IN FNT WORD 2 OF A TAPE FILE ENTRY CONTAINING SECONDARY DEVICE NUMBER (EST ORDINAL).

C.FTL (0)
FIRST BYTE OF A TWO-BYTE FIELD IN FNT WORD 3 CONTAINING THE TIME LIMIT FOR A JOB IN THE INPUT QUEUE.

C.FTYPE (3)
BYTE IN FNT WORD 1 CONTAINING FILE TYPE FIELD.

C.HEC (4)
BYTE OF WORD P.HEC CONTAINING THE HARDWARE ERROR COUNT.

C.ICEBUF (0)
BYTES IN CMR WHICH CONTAIN ADDRESS AND LENGTH OF THE ICEBOX I/O BUFFER.

C.LRD (3)
BYTES IN CMR WHICH CONTAIN ADDRESS AND LENGTH OF THE LOGICAL RECORD DEFINITION TABLE.

C.NCP (4)
BYTE OF WORD P.NCP CONTAINING THE NUMBER OF CONTROL POINTS.

C.NOVA (1)
BYTE OF WORD P.NOVA CONTAINING THE FWA/10B OF THE NOVA TABLE (GP250).

C.NOVAL (2)
BYTE OF WORD P.NOVA CONTAINING THE LENGTH OF THE NOVA TABLE.

C.PFMCH (4)
PERMANENT FILE INTERLOCK BYTE IN WORD P.PFM1.

C.PPFWA (1000)
LOCATION IN PP MEMORY AT WHICH PP RESIDENT IS TO BEGIN EXECUTION OF A PRIMARY OVERLAY. USED BOTH AS THE SECOND PARAMETER ON IDENT CARD AND IN ADDRESS FIELD OF ORG ON ALL PRIMARY PP OVERLAYS.

C.PPSWA (2000)
LOCATION IN PP MEMORY FOR BEGINNING EXECUTION OF A SECONDARY OVERLAY. USED SAME AS C.PPFWA.

C.PPTWA (3000)
LOCATION IN PP MEMORY AT WHICH TO BEGIN EXECUTION OF A TERTIARY OVERLAY. USED SAME AS C.PPFWA.

C.PP4WA (4000)
LOCATION IN PP MEMORY AT WHICH TO BEGIN EXECUTION OF A FOURTH LEVEL OVERLAY.

C.PP5WA (5000)
LOCATION IN PP MEMORY AT WHICH TO BEGIN EXECUTION OF A FIFTH LEVEL OVERLAY.

C.PP6WA (6000)
LOCATION IN PP MEMORY FOR BEGINNING EXECUTION OF A SIXTH LEVEL OVERLAY.

C.PP7WA (7000)
LOCATION IN PP MEMORY AT WHICH TO BEGIN EXECUTION OF A SEVENTH LEVEL OVERLAY.

C.RBRA (3)
BYTE IN RBR HEADER CONTAINING PERMISSIBLE ALLOCATION.

C.RBRAD (0)
BYTE IN WORD P.RBR CONTAINING FWA OF THE RBR AREA.

C.RBRLAV (3)
BYTE IN SECOND WORD OF EACH RBR HEADER CONTAINING COUNT OF UNASSIGNED RECORD BLOCKS DEFINED IN THE RBR.

C.RBRTPA (0)
BYTE IN FIRST WORD OF EACH RBR HEADER DEFINING DEVICE TYPE REFERENCED.

C.RBRUNT (1)
BYTE IN WORD ONE OF RBR HEADER CONTAINING DST ORDINAL.

C.RBTAL (2)
BYTE IN FIRST WORD OF RBT WORD PAIR CONTAINING ALLOCATION TYPE OF FILE.

C.RBTCL (1)
BYTE IN WORD P.PFM2 CONTAINING THE NUMBER OF PRU/8 IN RBTC.

C.RBTC1 (2)
C.RBTC2 (3)
C.RBTC3 (4)
THESE BYTES IN P.PFM2 CONTAIN THE CURRENT END-OF-INFORMATION POINTER FOR THE RBTC USED BY THE PERMANENT FILE SYSTEM.

C.RBTFB (1)
BYTE IN FIRST WORD OF RBT WORD PAIR CONTAINING BYTE NUMBER OF FIRST RECORD BLOCK ADDRESS.

C.RBTLPR (3)
BYTE IN SECOND WORD OF AN RBT WORD PAIR CONTAINING THE TERMINAL PRU NUMBER + 1 USED IN LAST EDITION OF A RANDOM FILE.

C.RBTLRB (4)
REFERENCES FOURTH BYTE OF CENTRAL MEMORY WORD IN RECORD BLOCK TABLE. DENOTES BYTE THAT CONTAINS LAST RECORD BLOCK IN PREVIOUS GENERATION.

C.RBTPRU (3)
BYTE IN FIRST RBT WORD ASSIGNED TO EACH FILE DEFINING LAST + 1 PRU ASSIGNED TO THAT FILE.

C.RBTRBR (1)
BYTE IN FIRST WORD OF RBT WORD PAIR CONTAINING RBR ORDINAL FOR THE FILE.

C.RBTWPL (0)
BYTE IN EACH RBT WORD PAIR CONTAINING LINKAGE TO NEXT RBT WORD PAIR FOR THAT FILE.

C.RQSCS (0)
BYTE IN STACK POINTER WORD(P.RQS) CONTAINING A COUNT OF THE STACK ENTRY WORD PAIRS.

C.RQSFS (2)
BYTE IN STACK POINTER WORD (P.RQS) CONTAINING ADDRESS OF FIRST STACK ENTRY.

C.RSTA (2)
BYTE IN M.EREQS MONITOR FUNCTION CONTAINING PP AVAILABLE FLAG.

C.RSTRA (1)
BYTE IN M.EREQS MONITOR FUNCTION CONTAINING REQUEST ACCEPTED FIELD WHICH IS USED FOR MONITOR-PP RESIDENT COMMUNICATION.

C.RSTU (4)
BYTE IN M.EREQS MONITOR FUNCTION CONTAINS UNIT NUMBER (DST ORDINAL).

C.RSTWP (3)
BYTE IN M.EREQS MONITOR FUNCTION CONTAINING REQUEST STACK WORD PAIR ADDRESS.

C.RWPPCC (3)
BYTE IN READP/WRITEP CONTROL WORD INTO WHICH STACK PROCESSOR FOR CONTROL PHASE 1 PLACES CHANNEL NUMBER TO BE USED IN DATA TRANSMISSION.

C.RWPPCF (0)
BYTE CONTAINING PHASE CONTROL FLAG IN CONTROL WORD FOR READP/ WRITEP. PHASES: 0 = REQUEST IN STACK, 1 = SET CHANNEL, 2 = CHANNEL SET, AWAIT TRANSMISSION, 3 = TRANSMISSION IN PROGRESS, 4 = ORDER COMPLETED.

C.RWPPLW (2)
BYTE IN READP/WRITEP CONTROL WORD CONTAINING LWA + 1 OF DATA TRANSMITTED. IT IS UPDATED BY PP RESIDENT DURING ORDER EACH TIME IT COMPLETES PHASE 3.

C.RWPPST (3)
IN READP/WRITEP CONTROL WORD, OPERATION STATUS AVAILABLE IN PHASE 4 IS CONTAINED IN THIS BYTE.

C.RWPPWC (4)
IN READP/WRITEP CONTROL WORD, WORD COUNT FOR TRANSMISSION DURING PHASE 3 IS CONTAINED IN THIS BYTE.

C.RWPPWT (1)
IN READP/WRITEP CONTROL WORD, TOTAL NUMBER OF WORDS TRANSMITTED DURING ALL PHASE 3'S IS CUMULATED BY PP RESIDENT IN THIS BYTE.

C.SDL (0)
BYTE IN WORD P.PFM2 CONTAINING THE NUMBER OF ENTRIES PER SUB-DIRECTORY.

C.SDT (2)
BYTE IN WORD P.PFM1 CONTAINING THE FWA OF THE SUB-DIRECTORY TABLE.

C.SDTL (0)
BYTE IN WORD P.PFM1 CONTAINING THE NUMBER OF PERMANENT FILE SUB-DIRECTORIES.

C.SEQ (2)
BYTE IN WORD P.SEQ CONTAINING THE FWA/10B OF THE SEQUENCER TABLE.

C.SEQL (3)
BYTE IN WORD P.SEQ CONTAINING THE LENGTH OF THE SEQUENCER TABLE.

C.STATCP (2)
BYTE IN WORD T.STATCP WHICH CONTAINS THE CURRENT STATUS OF THE CPUS (6 BITS STATUS FOR EACH CPU)
IF CPU STATUS = 40 CPU ON
IF CPU STATUS = 44 CPU ON AND DELEGATED
IF CPU STATUS = 60 CPU OFF
IF CPU STATUS = 77 CPU DOES NOT EXIST

C.STCPU (4)
BYTE IN WORD 1 OF STACK REQUEST CONTAINING CONTROL POINT AND UNIT NUMBER.

C.STEI (4)
EMPTY ENTRY INDICATOR IN WORD 1 OF A STACK ENTRY. IF 0, ENTRY IS NOT IN USE.

C.STFB (3)
FLAG BYTE IN WORD 2 OF A STACK REQUEST.

C.STO (3)
SPECIFIC ORDER IN WORD 1 OF A STACK REQUEST: HIGH ORDER 6 BITS ARE A RECORD LEVEL NUMBER WHEN RELEVANT (ORDER = O.SKF).

C.STPFW (2)
NEXT ADDRESS IN PP MEMORY FOR DATA IN WORD 2 OF READP/WRITEP STACK REQUEST. USED BY PP RESIDENT TO COMPUTE BYTE COUNT AND AS FWA FOR DATA TRANSMISSION IN A CALL TO R.READPP OR R.WRITEP.

C.STPLW (4)
LAST ADDRESS IN PP MEMORY FOR DATA IN WORD 2 OF A READP/WRITEP STACK REQUEST. USED BY PP RESIDENT TO COMUPTE BYTE COUNT IN CALL TO R.READP OR R.WRITEP.

C.STPMS (1)
LOCATION OF MESSAGE BUFFER OF PP IN WORD 2 OF A READP/WRITEP STACK REQUEST. FIRST 3 WORDS ARE USED FOR COMMUNICATION WITH STACK PROCESSOR.

C.STPPRU (2)
PRU NUMBER AT WHICH TO BEGIN DATA TRANSMISSION IN WORD 1 OF A STACK REQUEST WITH FLAG SET FOR NO FNT.

C.STPRBA (0)
RBT ADDRESS OF WORD PAIR CONTAINING RECORD BLOCK AT WHICH TO BEGIN DATA TRANSMISSION IN WORD 1 OF STACK REQUEST WITH FLAG SET FOR NO FNT.

C.STPRBN (1)
RBT ORDINAL OF RECORD BLOCK AT WHICH TO BEGIN DATA TRANSMISSION IN WORD 1 OF STACK REQUEST WITH FLAG SET FOR NO FNT.

C.STPWC (0)
COUNT OF BYTES TO BE TRANSMITTED IN WORD 1 OF READP/WRITEP STACK REQUEST.

D.BA (40)
D.BA THROUGH D.BA+4 CONTAIN FIRST WORD OF FILE ENVIRONMENT TABLE (FET) LOCATED BY RELATIVE ADDRESS IN LOW ORDER 18 BITS OF INPUT REGISTER.

D.CPAD (74)
TYPICALLY CONTAINS ADDRESS OF CONTROL POINT AREA CURRENLTY IN USE BY PP. A PRIMARY OVERLAY USUALLY STORES THE ADDRESS AS PART OF ITS INITIALIZATION.

D.DTS (37)
: HIGH ORDER 6 BITS OF D.DTS CONTAIN DEVICE TYPE FOUND IN HIGH ORDER PORTION OF BYTE 0 OF SECOND WORD OF FNT. LOW ORDER 6 BITS OF D.DTS CONTAIN ALLOCATION TYPE FOUND IN LOW ORDER PORTION OF BYTE 0 OF SECOND WORD OF FET.

D.EST (32)
: D.EST THROUGH D.EST+4 CONTAIN EST ENTRY IN PROCESS.

D.FA (57)
: CONTAINS ADDRESS OF SECOND WORD OF FNT ENTRY IN PROCESS.

D.FIRST (60)
: D.FIRST AND D.FIRST+1 CONTAIN 18-BIT CM ADDRESS SPECIFYING BEGINNING OF A CIRCULAR BUFFER (CONTENTS OF FIRST POINTER (WORD 2) FROM A FET).

D.FL (55)
: CENTRAL MEMORY FIELD LENGTH/100B OF CONTROL POINT TO WHICH PP IS CURRENTLY ATTACHED. PRIMARY OVERLAY USUALLY STORES FIELD LENGTH AS PART OF INITIALIZATION.

D.FNT (20)
: D.FNT THROUGH D.FNT+9 CONTAIN WORDS 2 AND 3 OF FNT ENTRY FOR FILE IN PROCESS. WORDS 2 AND 3 ARE REFERRED ALTERNATELY TO AS THE FST ENTRY.

D.HN (71)
: CONSTANT 100B. GENERALLY, D.HN IS PRESET BY A PRIMARY OVERLAY FOR USE BY A SECONDARY OVERLAY.

D.IN (62)
: D.IN AND D.IN+1 CONTAIN FET.

D.JECS (45)
: USED BY 2TJ TO RETURN ECS FIELD LENGTH REQUIREMENT TO CALLING PROGRAM.

D.JFL (37)
: USED BY 2TJ TO RETURN CM FIELD LENGTH REQUIREMENT TO CALLING PROGRAM.

D.JPR (46)
: USED BY 2TJ TO RETURN COMPUTED PRIORITY TO CALLING PROGRAM.

D.LIMIT (66)
: D.LIMIT AND D.LIMIT+1 CONTAIN 18-BIT ADDRESS SPECIFYING LWA + 1 OF CIRCULAR BUFFER (CONTENTS OF LIMIT POINTER (WORD 5) FROM FET).

D.OUT (64)
: D.OUT AND D.OUT+1 CONTAIN OUT POINTER (WORD 4) FROM FET.

D.PPIR (75)
CONTAINS CM ADDRESS OF PP INPUT REGISTER. INITIALIZED AT DEADSTART TIME AND MUST NEVER BE ALTERED.

D.PPIRB (50)
D.PPIRB THROUGH D.PPIRB+4 HOLD THE CONTENTS OF PP INPUT REGISTER. PRIMARY OVERLAY USUALLY STORES INPUT REGISTER CONTENTS AS PART OF INITIALIZATION.

D.PPMES1 (77)
CONTAINS CM ADDRESS OF FIRST WORD OF PP MESSAGE BUFFER. INITIALIZED AT DEADSTART TIME AND MUST NEVER BE ALTERED.

D.PPONE (70)
CONTAINS CONSTANT 1. GENERALLY, D.PPONE IS PRESET BY A PRIMARY OVERLAY FOR USE BY A SECONDARY OVERLAY.

D.PPOR (76)
CONTAINS CM ADDRESS OF PP OUTPUT REGISTER. INITIALIZED AT DEADSTART TIME AND MUST NEVER BE ALTERED.

D.RA (55)
CONTAINS CENTRAL MEMORY REFERENCE ADDRESS/100B OF CONTROL POINT TO WHICH PP IS ATTACHED. PRIMARY OVERLAY USUALLY STORES ADDRESS AS PART OF INITIALIZATION.

D.TH (72)
CONTAINS CONSTANT 1000/8. GENERALLY, D.TR IS PRESET BY A PRIMARY OVERLAY FOR USE BY A SECONDARY OVERLAY.

D.TR (73)
CONTAINS CONSTANT 3. GENERALLY, D.TR IS PRESET BY A PRIMARY OVERLAY FOR USE BY A SECONDARY OVERLAY.

F.ERAR (2)
VALUE OF ERROR FLAG SET FOR CP ARITHMETIC ERROR ABORT. SENSED BY MTR; ERROR MESSAGE IS WRITTEN BY 1EJ.

F.ERCP (4)
VALUE OF ERROR FLAG SET FOR CP ABORT. F.ERCP USED IF CP PROGRAM ABORTS EXECUTION; PROGRAM MUST WRITE A MESSAGE TO DAYFILE.

F.ERECP (12)
VALUE OF ERROR FLAG SET WHEN MTR DETECTS PERMANENT PARITY IN ECS DURING ECS STORAGE MOVE. 1EJ WRITES A MESSAGE TO DAYFILE.

F.EREX (11)
VALUE OF ERROR FLAG SET BY 1AJ WHEN IT DETECTS A CONTROL CARD ERROR, OR BY MTR WHEN IT GETS AN ABT REQUEST WITH THE EXIT(S) BIT (BIT 36) SET.

F.ERHANG (16)
ERROR FLAG FOR JOB HUNG IN AUTOMATIC RECALL.

F.ERJC (13)
ERROR FLAG FOR A JOB CARD ERROR.

F.ERK (7)
VALUE OF ERROR FLAG IN CONTROL POINT AREA SET WHEN OPERATOR TYPES ≠N.KILL≠. THERE WILL BE NO OUTPUT FROM THIS JOB.

F.EROD (6)
VALUE OF ERROR FLAG SET FOR OPERATOR DROP TYPE-IN. 1EJ WRITES A MESSAGE TO DAYFILE.

F.ERPA (14)
ERROR FLAG FOR A JOB WHICH HAS BEEN PRE-ABORTED,I.E., BEFORE IT COMES TO CONTROL POINT.

F.ERPCE (5)
VALUE OF ERROR FLAG SET FOR PP CALL ERROR ABORT. SENSED BY MTR; ERROR MESSAGE IS WRITTEN BY 1EJ. USED WHEN CENTRAL PROGRAM REQUESTS PP PROGRAM WITH NAME THAT DOES NOT BEGIN WITH A LETTER.

F.ERRCL (15)
ERROR FLAG FOR A BAD PP CALL WITH THE AUTO-RECALL BIT SET.

F.ERRN (10)
VALUE OF ERROR FLAG IN CONTROL POINT AREA SET WHEN THE OPERATOR TYPES ≠N.RERUN≠. THIS JOB WILL BE PUT BACK INTO THE INPUT QUEUE.

F.ERPP (3)
VALUE OF ERROR FLAG SET FOR PP ABORT. PP REQUESTING ABORT IS RESPONSIBLE FOR WRITING MESSAGE.

F.ERTL (1)
VALUE OF ERROR FLAG SET FOR CP TIME LIMIT ABORT. SENSED BY MTR; ERROR MESSAGE IS WRITTEN BY 1EJ.

LE.FNT (3)
NUMBER OF CENTRAL MEMORY WORDS IN ONE FNT ENTRY.

L.CPNUM (7)
MASK OF ONES EQUAL TO THE LENGTH IN BITS OF HIGHEST NUMBERED CONTROL POINT.

L.PPHDR (5)
NUMBER OF PP WORDS COMPRISING HEADER INFORMATION APPENDED BY ASSEMBLER. LOADING OF ALL PP OVERLAYS BEGINS AT C.PPXWA MINUS L.PPHDR.

M.ABORT *
(13B-ABORT CONTROL POINT)

M.CCPA *
(35B-CHANGE CONTROL POINT ASSIGNMENT)

M.CPUST *
(36B-CHANGE CPU STATUS)

M.DCP *
(16B-DROP CENTRAL PROCESSOR)

M.DEQP *
(23B-DROP EQUIPMENT)

M.DFM *
(01-PROCESS DAYFILE MESSAGE)

M.DPP *
(12B-DROP PP)

M.DTAPE *
(32B-TURN EQUIPMENT OFF)

M.EREQS *
(34B-ENTER REQUEST STACK)

M.ESTZ *
(33B-EST Z BYTE UPDATE)

M.ICE *
(06-INITIATE CENTRAL EXECUTIVE)

M.NTIME *
(14B-TIME LIMIT)

M.OPDROP *
(30B-OPERATOR DROP)

M.PAUSE *
(17B-PAUSE FOR RELOCATION)

M.PPTIME *
(04-ASSIGN PP TIME)

----
* SEE CHAPTER 5.4

M.RACT *
(27B=REQUEST CONTROL POINT ACTIVITY)

M.RBTSTO *
(07B= REDUCE RBT STORAGE)

M.RCH *
(02=REQUEST CHANNEL)

M.RCLCP *
(21B=RECALL CENTRAL PROGRAM)

M.RCP *
(15B=REQUEST CENTRAL PROCESSOR)

M.REM *
(25B=ASSIGN ERROR EXIT MODE)

M.REQP *
(22B=REQUEST EQUIPMENT)

M.RPJ *
(37B=REQUEST PERIPHERAL JOB)

M.RPP *
(20B=REQUEST PP)

M.RPRI *
(27B=REQUEST PRIORITY)

M.RSTOR *
(10B=REQUEST STORAGE)

M.RTAPE *
(31B=TURN EQUIPMENT ON)

M.SEF *
(30B=SET ERROR FLAG)

M.SEQ *
(26B=ASSIGN JOB SEQUENCE NUMBER)

M.STEP *
(05=MONITOR STEP CONTROL)

----
* SEE CHAPTER 5.4

O.BPRU (16)
BACKSPACE N PRU≠S. NUMBER OF PRU≠S TO BE BACKSPACED IS GIVEN IN THIRD BYTE OF SECOND WORD OF THE ORDER. O.BPRU REQUESTS REPOSITIONING DEFINED BY PHYSICAL RATHER THAN LOGICAL UNITS. NO DATA IS TRANSMITTED.

O.RCHN (17)
RELEASE CHAIN. ALL RECORD BLOCKS ASSIGNED TO A FILE AND THE RBT WORD PAIRS CONTAINING THEM ARE RELEASED. FNT IS RESET TO AN EMPTY CONDITION IF ITS ADDRESS IS SUPPLIED IN THE ORDER. REQUESTS 16 AND 17 REQUIRE NO COMMUNICATION WITH THE DEVICE AND, THEREFORE, ARE GIVEN HIGHEST PRIORITY IN THE SEARCH FOR THE NEXT ORDER TO BE EXECUTED. ALL OTHER REQUESTS ARE ASSIGNED PRIORITY BASED ON REPOSITIONING REQUIREMENTS.

O.RCMPR (2)
READ INTO CENTRAL MEMORY AFTER DROPPING FIRST THREE CM WORDS OF FIRST PRU. USED BY STITCH FOR LOADING PROGRAM FOR SYSTEM LIBRARY ELIMINATING THE THREE WORD HEADER ADDED TO SYSTEM PROGRAMS BY EDITLIB.

O.RDNS (3)
VALUE OF THE STACK PROCESSOR ORDER CODE FOR READNS.

O.RDP (10)
READ INTO REQUESTING PP≠S MEMORY UNTIL A SHORT PRU IS ENCOUNTERED OR UNTIL INPUT AREA IS FULL.

O.RDPNP (11)
READ INTO REQUESTING PP AFTER DROPPING FIRST THREE CM WORDS OF FIRST PRU. USED FOR ALL PP SYSTEM PROGRAM CALLS.

O.RDSK (1)
READ INTO CENTRAL MEMORY UNTIL A SHORT PRU IS ENCOUNTERED OR UNTIL BUFFER IS FULL. SET FNT TO REFERENCE FIRST PRU FOLLOWING FIRST END-OF-RECORD OF LEVEL X OR GREATER. LEVEL IS GIVEN IN HIGH-ORDER 6 BITS OF THE ORDER BYTE.

O.READ (0)
READ INTO CENTRAL MEMORY UNTIL A SHORT PRU IS ENCOUNTERED OR BUFFER IS FULL (IN=OUT).

O.SKB (13)
SKIP BACKWARD N RECORDS OF LEVEL X OR GREATER. LEVEL IS SPECIFIED IN HIGH 6 BITS OF THE ORDER BYTE; NUMBER OF RECORDS TO BE SKIPPED IS GIVEN IN THIRD BYTE OF SECOND WORD OF THE ORDER. NO DATA IS TRANSMITTED.

O.SKF (12)
SKIP FORWARD N RECORDS OF LEVEL X OR GREATER. LEVEL IS SPECIFIED IN HIGH 6 BITS OF THE ORDER BYTE; NUMBER OF RECORDS TO BE SKIPPED IS GIVEN IN THIRD BYTE OF SECOND WORD OF THE ORDER. NO DATA IS TRANSMITTED.

O.WRP (14)
WRITE FROM REQUESTING PP, FULL PRU≠S ONLY.

O.WRPR (15)
WRITE FROM REQUESTING PP, ENDING WITH A SHORT PRU OF LEVEL SPECIFIED IN HIGH ORDER 6 BITS OF ORDER BYTE. IF EOF FLAG BIT IS SET IN THIS ORDER, A ZERO LENGTH PRU OF LEVEL 17 IS WRITTEN FOLLOWING SHORT PRU TERMINATING RECORD.

O.WRT (4)
WRITE FULL PRU≠S FROM CENTRAL MEMORY.

O.WRTR (5)
WRITE FROM CENTRAL MEMORY, ENDING WITH A SHORT PRU OF LEVEL SPECIFIED IN HIGH ORDER 6 BITS OF THE ORDER BYTE. IF EOF FLAG BIT IS FOUND IN THIS ORDER, A ZERO LENGTH PRU OF LEVEL 17 IS WRITTEN FOLLOWING SHORT PRU TERMINATING RECORD.

P.CST (5)
WORD IN CMR CONTAINING CHANNEL STATUS TABLE POINTERS.

P.DFB (3)
ADDRESS OF DAYFILE BUFFER POINTER WORD. ONLY THE FIRST BYTE (BYTE 0) IS USED: IT CONTAINS CM ADDRESS/ 10B OF DAYFILE BUFFER.

P.ECST (11B)
WORD IN CMR CONTAINING POINTERS TO VARIOUS ECS TABLES.

P.EST (5)
ADDRESS OF EST POINTER WORD. BYTE 0 CONTAINS 12-BIT FIRST WORD ADDRESS: BYTE 1 CONTAINS 12-BIT LAST WORD ADDRESS PLUS ONE.

P.FNT (4)
ADDRESS OF FNT POINTER WORD. BYTE 0 CONTAINS 12-BIT FIRST WORD ADDRESS: BYTE 1 CONTAINS 12-BIT LAST WORD ADDRESS PLUS ONE.

P.HEC (4)
CMR LOCATION OF A WORD CONTAINING THE HARDWARE ERROR COUNT.

P.INS (10)
ADDRESS OF A POINTER WORD TO AN INSTALLATION AREA: CONTENT IS UNSPECIFIED.

P.LECST (12)
WORD IN CMR CONTAINING LENGTHS OF TABLES SPECIFIED BY P.ECST.

P.LIB (1)
ADDRESS OF LIBRARY DIRECTORY POINTER WORD. BYTES 0 AND 1 CONTAIN RIGHT JUSTIFIED 18-BIT FIRST WORD ADDRESS OF LIBRARY DIRECTORY. BYTES 2 AND 3 CONTAIN RIGHT JUSTIFIED 18-BIT LAST WORD ADDRESS PLUS ONE. BYTE 4 CONTAINS A DEADSTART LOAD FLAG: IT MUST ALWAYS BE ZERO WHEN A DISK OR RECOVERY DEADSTART IS ATTEMPTED.

P.NCP (5)
CMR LOCATION OF A WORD CONTAINING THE NUMBER OF CONTROL POINTS.

P.NOVA (3)
CMR LOCATION OF THE NOVA POINTER WORD.

P.PFM1 (6)
P.PFM2 (7)
CMR LOCATIONS OF TWO POINTER WORDS USED BY THE PERMANENT FILE MANAGER.

P.RBR (2)
ADDRESS OF RBR POINTER WORD. (ALSO SERVES AS RBT POINTER WORD -SEE P.RBT.) BYTES 0 AND 1 CONTAIN RIGHT JUSTIFIED 18-BIT FIRST WORD ADDRESS OF RBR TABLE AREA.

P.RBT (2)
ADDRESS OF RBT POINTER WORD. (ALSO SERVES AS RBR POINTER WORD- SEE P.RBR.) BYTE 2 CONTAINS FIRST WROD ADDRESS/2 OF RBT EMPTY CHAIN. BYTE 3 CONTAINS CURRENT LENGTH/100B ENTIRE RBT AREA. BYTE 4 CONTAINS (LWA + 1)/100B OF CENTRAL MEMORY.

P.RQS (13)
ADDRESS OF REQUEST STACK AREA POINTER WORD. BYTE 0 CONTAINS STACK ENTRY WORD PAIR COUNT. BYTE 2 CONTAINS FIRST WORD ADDRESS/2 OF ACTUAL REQUEST STACK. BYTE 3 CONTAINS NUMBER OF ALLOCATABLE DEVICES (N.DEVICE). BYTE 4 CONTAINS FWA/10B OF DST ENTRIES. ALL DST ENTRIES APPEAR AT BEGINNING OF REQUEST STACK AREA' FOLLOWED IMMEDIATELY BY ACTUAL REQUEST STACK.

P.SEQ (4)
WORD IN CMR CONTAINING DIAGNOSTIC SEQUENCER POINTERS.

P.ZERO (0)
ADDRESS OF CENTRAL MEMORY WORD CONTAINING ALL ZEROS. USED BY MOST PP ROUTINES AS A QUICK MEANS OF ZEROING FIVE SUCCESSIVE PP LOCATIONS. THE SYSTEM IS DESTROYED BY SETTING CONTENTS OF P.ZERO TO NONZERO.

R.CPFL (627)
LOCATION WITHIN PP RESIDENT CONTAINING FIELD LENGTH/100B OF CONTROL POINT TO WHICH PP IS ATTACHED. R.CPFL IS RESET EACH TIME R.PAUSE ROUTINE IS ENTERED.

R.CPRA (631)
LOCATION IN PP RESIDENT CONTAINING REFERENCE ADDRESS/100B OF CONTROL POINT TO WHICH PP IS ATTACHED. R.CPRA IS RESET EACH TIME EACH R.PAUSE ROUTINE IS ENTERED.

R.DCH (714)

CALLING SEQUENCE:	LOAD	CHANNEL NUMBER
	RJM	R.DCH
	R.DCH	WILL CAUSE THE SPECIFIED CHANNEL TO BE DROPPED.

R.DFM (650)

CALLING SEQUENCE: LOAD L(MESSAGE)+FLAG BITS
RJM R.DFM

MESSAGE FROM PP MEMORY IS WRITTEN TO DAYFILE AND/OR CONSOLE. FLAG BITS ARE CONTAINED IN HIGH ORDER 6-BITS OF A REGISTER UPON ENTRY TO R.DFM; THEY DETERMINE MESSAGE DESTINATIONS. FLAG BIT VALUES ARE GIVEN BELOW; ONE OR MORE BITS MAY BE ON; ALL ARE OPTIONAL.

1 DAYFILE ONLY (A DISPLAY)
2 CONTROL POINT 0 (SYSTEM) MESSAGE
4 NO A DISPLAY

R.EREQS (300)

CALLING SEQUENCE: STORE L(REQUESTS) IN D.T0
RJM R.EREQS

ADDS THE CONTROL POINT NUMBER TO THE ALREADY FORMATTED REQUEST AND SEARCHES THE CENTRAL MEMORY REQUEST STACK FOR AN EMPTY ENTRY. THE MONITOR FUNCTION, M.EREQS, IS CALLED AND PP RESIDENT ITERATES UNTIL THE MONITOR ACCEPTS THE REQUEST. IF THE AVAILABLE FLAG (BIT S.STF+S.STFA OF BYTE C.STFB OF THE SECOND WORD OF THE REQUEST) IS SET,R.EREQS EXITS TO R.IDLE. OTHERWISE IT RETURNS CONTROL TO ITS CALLER.

R.EREQSP (400B)

PP RESIDENT LOCATION MODIFIED BY THE PERMANENT FILE MANAGER FOR DIRECTORY SEARCH.

R.IDLE (100)

CALLING SEQUENCE: LJM R.IDLE

IDLE LOOP; PP RESIDENT CONTINUALLY SCANS ITS INPUT REGISTER FOR SOMETHING TO DO.

R.MTR (450)

CALLING SEQUENCE: STORE FUNCTION PARAMETERS IN D.T1 TO D.T4
LOAD FUNCTION CODE
RJM R.MTR

PLACES THE FUNCTION CODE IN D.T0, WRITES D.T0 THROUGH D.T4 TO THE OUTPUT REGISTER AND WAITS FOR THE OUTPUT REGISTER.

R.OVL (124)

CALLING SEQUENCE: LOAD A REGISTER LOAD ADDRESS
RJM R.OVL

CAUSES AN OVERLAY WHOSE NAME APPEARS IN D.T6 AND D.T7 (LEFT JUSTIFIED) TO BE LOADED INTO THE PP BEGINNING AT THE ADDRESS SPECIFIED IN THE A REGISTER. R.OVL IS USED BY PP OVERLAYS TO LOAD HIGHER LEVEL OVERLAYS AND BY PP RESIDENT TO LOAD THE OVERLAY NAMED IN THE INPUT REGISTER. PP RESIDENT DOES NOT REFERENCE THE DISK DIRECTLY TO LOAD DISK RESIDENT OVERLAYS BUT MAKES A CALL TO THE STACK PROCESSOR.

R.OVLJ (112)

CALLING SEQUENCE: STORE NAME OF OVERLAY IN D.T6, D.T7
LJM R.OVLJ

GO TO R.OVLJ TO LOAD A NEW PRIMARY OVERLAY AND TRANSFER CONTROL TO IT.

R.PAUSE (430)

```
CALLING SEQUENCE:  RJM     R.PAUSE
                   STD     D.RA
```

EXITS IF PP IS ATTACHED TO CONTROL POINT ZERO OR IF STORAGE MOVE FLAG IS NOT SET. OTHERWISE, THE MONITOR FUNCTION M.PAUSE IS IS ISSUED AND PP PAUSES UNTIL MONITOR HAS COMPLETED STORAGE MOVE FOR THAT CONTROL POINT. IN ANY EVENT, BEFORE AN EXIT IS MADE FROM R.PAUSE, THE FOLLOWING INFORMATION IS SET:

```
(D.TO + C.CPSTAT)      CONTROL POINT STATUS
(D.TO + C.CPEF)        CONTROL POINT ERROR FLAG
(D.TO + C.CPRA)        CONTROL POINT RA (HUNDREDS)
(D.TO + C.CPFL)        CONTROL POINT FL (HUNDREDS)
A REGISTER             CONTROL POINT RA
```

D.RA AND D.FL (IF SIGNIFICANT) SHOULD ALWAYS BE RESET AFTER A JUMP TO R.PAUSE.

R.PROCES (450)

IDENTICAL WITH R.MTR.

R.RCH (704)

```
CALLING SEQUENCE:  LOAD    CHANNEL NUMBER
                   RJM     R.RCH
```

CHANNEL NUMBERS IN A REGISTER ARE STORED IN BYTE D.T1, MONITOR FUNCTION M.RCH INSERTED IN D.TO, AND D.TO-D.T4 WRITTEN TO OUTPUT REGISTER FOR THAT PP. CHANNELS ARE ASSIGNED BY MTR ON THE FOLLOWING PRIORITY BASIS:
IF ALTERNATE CHANNELS ARE SPECIFIED, MTR STOPS LOOKING FOR ALTERNATE CHANNELS UPON SENSING 6 BITS OF ZERO. THUS, IF ONE ALTERNATE CHANNEL IS DESIRED, THE PROGRAMMER MUST CLEAR D.T2 BEFORE ENTERING R.RCH SO THE SEARCH TERMINATES AT THAT POINT.
PROCEDURE FOR REQUESTING CHANNEL 12 WITH ALTERNATE CHANNEL 13:

```
LDN          0
STD          D.T2
LDC          1312B
RJM          R.RCH
```

MONITOR WILL STOP LOOKING FOR ALTERNATE CHANNELS AFTER FOUR CHANNELS HAVE BEEN INVESTIGATED.
WHEN R.RCH IS USED, D.T4 IS AUTOMATICALLY SET NONZERO; THE FUNCTION IS NOT CONSIDERED COMPLETE (OUTPUT REGISTER IS NOT CLEARED) UNTIL A CHANNEL CAN BE ASSIGNED. WHEN COMPLETE, BYTE 0 OF OUTPUT REGISTER IS CLEARED AND BYTE 4 IS SET NON-ZERO.
A CHANNEL REQUEST MAY BE MADE DIRECTLY TO MONITOR (M.RCH). ONE OTHER OPTION IS ALLOWED IN THIS CASE: MONITOR CANNOT ASSIGN THE CHANNEL, IT MOVES BYTE 4 OF OUTPUT REGISTER TO BYTE 0. R.RCH IN BYTE 4 OF OUTPUT REGISTER TO WAIT UNTIL THE CHANNEL IS ASSIGNED.

```
R.READP (R.WRITEP)   (460(470))
    CALLING SEQUENCE:  LOAD     L(REQUEST)
                       RJM      R.READP(R.WRITEP)
    COMPUTES PP WORD COUNT FROM FIRST AND LAST WORD ADDRESSES
    GIVEN IN ALREADY FORMATTED REQUEST AND ADDS COMPUTED WORD COUNT,
    ADDRESS OF PP MESSAGE BUFFER, AND CONTROL POINT NUMBER TO
    REQUEST.  REQUEST IS ENTERED IN STACK AND DATA IS TRANSMITTED
    VIA CHANNEL DIRECTLY TO(FROM) PP MEMORY.  UPON EXIT FROM
    PP MEMORY.  UPON EXIT FROM R.READP(R.WRITEP), THE FOLLOWING
    INFORMATION IS SET:
          (D.T3 + C.RWPPLW)        LWA + 1 OF DATA TRANSMITTED
          (D.T3 + C.RWPPST)        STATUS
          (D.T3 + C.RWPPWT)        NUMBER OF PP WORDS TRANSMITTED

R.RWIOP (547)
    PP RESIDENT LOCATION MODIFIED BY THE PERMANENT FILE MANAGER
    FOR DIRECTORY SEARCH.

R.RWP  (505)
    SPECIAL ENTRY POINT TO R.READP USED BY LDR.

R.RWPP  (530)
    WORD IN  R.READP MODIFIED BY LDR.

R.STB  (620)
    CALLING SEQUENCE:  LOAD     L(LIST)
                       RJM      R.STB
       WHERE LIST HAS THE FORM:
          L(BYTE)
          L(WORD 1)
          L (WORD 2)
               .
               .
               .
               .
          L(WORD N)
          ZERO
    AN ENTRY POINT TO R.STB CALLED R.STBMSK IS THE ADDRESS OF THE
    MASK ≠ANDED≠ WITH EACH WORD IN THE LIST BEFORE THE WORD IS
    ≠EXCLUSIVE ORED≠ WITH THE BYTE.  THIS MASK IS INITIALLY 7700B
    AND THIS VALUE SHOULD BE RESTORED BY ANY ROUTINE WHICH
    SUBSTITUTES AN ALTERNATE MASK.  R.STB IS USED PRIMARILY TO
    SUBSTITUTE CHANNEL NUMBERS IN DRIVER OVERLAYS.

R.STBMSK  (611)
    ADDRESS WITHIN PP RESIDENT OF A LOCATION USED BY R.STB ROUTINE.
```

R.TFL (634)

CALLING SEQUENCE: LOAD RELATIVE ADDRESS
RJM R.TFL

INSURES THAT A RELATIVE ADDRESS IS WITHIN FIELD LENGTH; 18-BIT ADDRESS IS ADDED TO CONTROL POINT REFERENCE ADDRESS (RA) AND COMPARED WITH FIELD LENGTH. IF ADDRESS IS OUT OF RANGE, R.TFL EXITS WITH A NEGATIVE A REGISTER; IF ADDRESS IS LEGAL. A REGISTER CONTAINS ABSOLUTE CM ADDRESS (RA + RELATIVE ADDRESS) UPON EXIT. CONTROL POINT RA AND FL ARE KEPT LOCALLY WITHIN PP RESIDENT AT R.CPRA AND R.CPFL, RESPECTIVELY; THESE LOCATIONS ARE RESET WHEN AN ENTRY TO R.PAUSE IS MADE.

R.WAIT (410)

CALLING SEQUENCE: RJM R.WAIT

PP IDLES UNTIL BYTE 0 OF OUTPUT REGISTER IS CLEAR.

R.WRITEP (470)

SEE R.READP.

S.CPA

RIGHT OFFSET OF RERUN BIT IN BYTE C.CPFP OF THE CONTROL POINT AREA.

S.CPDP (6)

RIGHT OFFSET OF PRIVATE PACK BIT IN BYTE C.CPFP OF THE CONTROL POINT AREA.

S.CPE (13)

RIGHT OFFSET OF THE EXPORT/IMPORT BIT IN BYTE C.CPFP OF WORD W.CPERT OF THE CONTROL POINT AREA.

S.CPF (0)

RIGHT OFFSET OF THE REPROCESS CONTROL CARD BIT IN BY C.CPFP OF WORD W.CPERT OF THE CONTROL POINT AREA.

S.CPG (1)

RIGHT OFFSET OF THE ABORT BIT IN BYTE C.CPFP OF THE CONTROL POINT AREA. THIS BIT IS SET BY 1AJ IF A CONTROL CARD ERROR OCCURS OR IF THE USER TRIES TO LOAD THE BINARY OUTPUT OF AN ASSEMBLY OR COMPILATION WHICH HAD ERRORS.

S.CPN (4)

RIGHT OFFSET OF THE CHECKPOINT BIT IN BYTE C.CPFP OF THE CONTROL POINT AREA. THIS BIT IS SET IF A VALID CHECKPOINT HAS BEEN TAKEN.

S.CPR (12)

RIGHT OFFSET OF THE RESPOND BIT IN BYTE C.CPFP OF THE CONTROL POINT AREA. THIS BIT IS SET FOR ALL RESPOND JOBS.

S.CPS (3)

RIGHT OFFSET OF THE SEQUENCER BIT IN BYTE C.CPFP OF THE CONTROL POINT AREA. BIT IS SET TO INDICATE INPUT FILE IS NOT TO PURGE.

S.CPX (5)
RIGHT OFFSET OF THE EXIT BIT IN BYTE C.CPFP OF THE CONTROL POINT AREA. THIS BIT IS SET WHENEVER 1AJ ENCOUNTERS AN EXIT CARD.

S.DIRPR (8)
NUMBER OF BIT POSITIONS TO RIGHT SHIFT IN PP WORD TO RIGHT JUSTIFY PROGRAM RESIDENCE TO BIT ZERO.

S.DIRPT (4)
NUMBER OF BIT POSITIONS TO RIGHT SHIFT IN A PP WORD TO RIGHT JUSTIFY THE PROGRAM TYPE TO BIT ZERO.

S.FCB (6)
BIT POSITION OF COMMON FILE CHANGE BIT IN FNT BYTE C.FCB.

S.FNTEQP (6)
RIGHT OFFSET OF EQUIPMENT CODE FIELD IN FNT. EQUIPMENT CODE FIELD IS POSITIONED IN BITS 6-11 OF BYTE ZERO.

S.FNTLK (5)
RIGHT OFFSET OF LOCK BIT IN FNT. LOCK BIT IS POSITIONED IN BIT 5 OF BYTE 3.

S.FNTTYP (3)
RIGHT OFFSET OF FILE TYPE FIELD IN FNT. FILE TYPE FIELD IS POSITIONED IN BITS 3-4 OF BYTE 3.

S.RBRUNT (6)
RIGHT OFFSET OF DST ORDINAL FIELD IN RBR HEADER.

S.RBTNEW (8)
RIGHT OFFSET OF NEW RBT FLAG BIT IN FLAG FIELD OF RBT WORD PAIR.

S.RBTRBR (3)
RIGHT OFFSET OF RBR ORDINAL FIELD IN RBT WORD PAIR.

S.RBTREL (7)
RIGHT OFFSET OF RELEASE BIT IN FLAG FIELD OF RBT WORD PAIR.

S.RBTRND (6)
RIGHT OFFSET OF RANDOM BIT IN FLAG FIELD OF RBT WORD PAIR.

S.STF (6)
RIGHT OFFSET OF FLAG FIELD IN A STACK REQUEST.

S.STFA (0)
RIGHT OFFSET (IN THE FLAG FIELD) OF PP-AVAILABLE BIT IN A STACK REQUEST.

S.STFEOF (4)
RIGHT OFFSET (IN FLAG FIELD) OF END-OF-FILE BIT IN STACK REQUEST.

S.STFETP (2)
RIGHT OFFSET (IN FLAG FIELD) OF FET-PRESENT BIT IN STACK REQUEST.

S.STFNTP (3)
RIGHT OFFSET (IN FLAG FIELD) OF FNT-PRESENT BIT IN STACK REQUEST.

S.STFPRI (5)
RIGHT OFFSET (IN FLAG FIELD) WHICH SPECIFIES HIGH PRIORITY FOR THIS STACK REQUEST.

S.STFREL (4)
RIGHT OFFSET (IN FLAG FIELD) OF RELEASE BIT IN STACK REQUEST.

S.STFXCT (1)
RIGHT OFFSET (WITHIN THE FLAG FILES) OF THE EXACT BIT IN BYTE C.STFB OF WORD W.STFB OF A REQUEST STACK ENTRY. OFFSET IS RELATIVE TO S.STF.

T.CLK (30)
LOCATION IN WHICH CLOCK IS KEPT, IN FORM *HH.MM.SS. T.CLK IS UPDATED BY MTR AND DISPLAYED ON TOP LINE OF LEFT SCOPE. TIME WILL BE TIME OF DAY, IF A TIME ENTRY TO DSD WAS MADE: OTHERWISE, IT WILL BE THE TIME SINCE DEADSTART.

T.CPA1 (200)
CONTROL POINT AREA ADDRESS OF CONTROL POINT 1.

T.CPJOBN (26)
CMR LOCATION OF A POINTER WORD CONTAINING THE JOB SEQUENCE NUMBER AND JOB COUNT.

T.CPT1 (56)
LOCATION OF THE CPU ACTIVITY STATUS. CPU A STATUS IS IN BYTE 4 CPU B STATUS IS IN BYTE 3. THE ACTIVITY STATUS IS THE CONTROL POINT AREA ADDRESS OF THE PROGRAM USING CURRENTLY THE CPU (0000 FOR IDLE).

T.DATE (31)
LOCATION OF TODAY≠S DATE IN THE FORM ENTERED BY OPERATOR AT DEADSTART TIME. DATE IS DISPLAYED ON TOP LINE OF LEFT SCOPE.

T.ECSPAR (57)

THIS WORD CONTAINS IN BYTES

0,1, ZERO,

2 AN ECS FLAW TABLE FULL FLAG. IT IS SET WHEN A FLAW COUNT OVERFLOW IS DETECTED DURING AN ECS TRANSFER (BY ICEBOX).

3 AN ECS PARITY FLAG WHICH CAN ASSUME VALUES OF 2 OR 4. IT IS SET WHEN A PARITY ERROR IS DETECTED DURING AN ECS STORAYE MOVE, AND INDICATES WHETHER 1 OR 2 CONTROL POINTS SHALL HAVE THEIR ERROR FLAGS SET TO F.ERECP. THE CONTROL POINT REQUESTING THE ECS WILL ALWAYS HAVE ITS ERROR FLAG SET. IN THE CASE OF STORAGE OVERLAP AND THE OCCURRENCE OF THE PARITY ERROR IN THE PART OF ECS WHICH BELONGS TO ANOTHER CONTROL POINT, THE FLAG WILL HAVE THE VALUE 4. AFTER THE FLAG IS SET, DSD WILL DISPLAY A MESSAGE STATING THAT AN ECS PARITY ERROR HAS OCCURRED AND MTR WILL NOT ASSIGN ECS ANYMORE. THESE CONDITIONS SHALL PREVAIL UNTIL THE NEXT DEADSTART.

4 THIS BYTE CONTAINS THE ADDRESS OF THE BLOCK IN ECS/1000B IN WHICH THE PARITY ERROR OCCURRED. DSD WILL DISPLAY THIS VALUE AS ABOVE.

T.JDATE (27)

LOCATION OF TODAY≠S DATE IN JULIAN FORMAT. VALUE IS COMPUTED FROM DATE ENTERED BY OPERATOR AT DEADSTART TIME.

T.MSC (40)

CONTAINS THE TIME TO THE MILLISECOND SINCE DEADSTART.

BYTE 1	TIME IN SECONDS (RESET EACH 2**12 SECONDS)
BYTE 2	MILLISECONDS SINCE UPDATING THE SECONDS
BYTES 3-4	TIME IN MSEC (RESET EACH 2**24 MSC)

T.MSP (37)

LOCATION OF MONITOR STEP FLAG USED FOR COMMUNICATION BETWEEN DSD AND MTR WHILE SYSTEM IS IN STEP MODE.

T.PPCX (1≤X≤9) (60,70,100,110,120,130,140,150,160)

SYMBOLS REPRESENT FIRST WORD ADDRESSES OF COMMUNICATION AREAS FOR PP≠S 1-9.

T.PPSX (X=1,2,...9,0) (41-52)

LOCATION OF THE PP STATUS WORD. BYTE 0 CONTAINS THE CONTROL POINT AREA ADDRESS OF THE CONTROL POINT TO WHICH THE PP IS ASSIGNED. T.PPS0(52) IS USED BY MTR AS A SCRATCH MEMORY WORD.

T.SLABX (1≤X≤6)

LOCATIONS CONTAINING SYSTEM LABEL DISPLAYED ON TOP LINE OF LEFT SCOPE.

T.STATCP (55)

WORD CONTAINING CURRENT STATUS OF CPU ON/OFF/DELEGATION. (SEE C.STATCP.)

T.UAS (56)
LOCATION OF UNASSIGNED STORAGE LENGTH. MTR KEEPS TALLY OF SIZE OF THE ≠HOLES≠ BETWEEN CONTROL POINTS IN T.UAS; 1RA USES THIS SIZE TO DETERMINE WHETHER OR NOT THERE IS ADEQUATE CENTRAL MEMORY TO BRING A GIVEN JOB TO A CONTROL POINT.

BYTE	0	CENTRAL MEMORY	UAS
BYTE	1	ECS	UAS

W.CKP (40)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING NUMBER OF CHECK-POINTS TAKEN FOR A JOB.

W.CPAR (157)
RELATIVE WORD IN A CONTROL POINT AREA WHICH CONTAINS THE AUTO-RECALL POINTER.

W.CPCAF (51)
RELATIVE FIRST WORD ADDRESS IN A CONTROL POINT AREA OF THE 100B WORD BUFFER CONTAINING CURRENT PRU OF CONTROL CARD STATEMENTS.

W.CPCAL (150)
RELATIVE LWA IN A CONTROL POINT AREA OF A 100B WORD BUFFER CONTAINING CURRENT PRU OF CONTROL CARD STATEMENTS.

W.CPCC (43)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING FLAGS RELEVANT TO THE LAST CONTROL CARD PROCESSED.

W.CPDFM (30)
FIRST OF SEVEN WORDS IN A CONTROL POINT AREA CONTAINING DAYFILE MESSAGE CURRENTLY ON THE B DISPLAY.

W.CPDFMC (155)
WORD IN THE CONTROL POINT AREA CONTAINING THE DAYFILE MESSAGE COUNT.

W.CPECS (22)
RELATIVE WORD IN A CONTROL POINT AREA CONTAINING ECS FIELD LENGTH/1000B AND ECS RA/1000B ASSIGNED TO A JOB.

W.CPEF (20)
RELATIVE WORD IN A CONTROL POINT AREA CONTAINING ERROR FLAG FOR JOB.

W.CPERT (40)
RELATIVE WORD IN A CONTROL POINT AREA CONTAINING INTERNAL FLAGS USED BY JOB PROCESSING ROUTINES.

W.CPFL (20)
RELATIVE WORD IN A CONTROL POINT AREA CONTAINING MEMORY FIELD LENGTH/100B ASSIGNED TO THE JOB.

W.CPFLAG (153)
RELATIVE WORD IN A CONTROL POINT AREA CONTAINING RERUN FLAGS.

W.CPINS (170)
RELATIVE FIRST WORD ADDRESS IN A CONTROL POINT AREA OF A 7 WORD ZONE RESERVED FOR INSTALLATION USE.

W.CPJCP (41)
WORD IN THE CONTROL POINT AREA CONTAINING JOB CARD INFORMATION FOR RERUN.

W.CPJNAM (21)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING SEVEN-CHARACTER JOB NAME.

W.CPLDR (152)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING VARIOUS LOADER FLAGS.

W.CPOAE (153)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING BYTE USED TO COMMUNICATE OPERATOR-ASSIGNED EQUIPMENT. MTR SETS EST ORDINAL OF EQUIPMENT REQUESTED BY DSD IN THIS BYTE FOR SUBSEQUENT TESTING BY REQ.

W.CPOUT (153)
RELATIVE WORD IN CONTROL POINT AREA WHICH CONTAINS AN OUTPUT FILE FLAG IN BYTE C.CPOUT.

W.CPPF1 (160)
W.CPPF2 (161)
WORDS IN THE CONTROL POINT AREA USED BY THE PERMANENT FILE MANAGER.

W.CPPRI (22)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING CURRENT JOB PRIORITY.

W.CPRES1 (156)
RELATIVE WORD IN CONTROL POINT AREA USED BY RESPOND.

W.CPRO
RELATIVE WORD IN A CONTROL POINT AREA CONTAINING THE ROLL-OUT FLAG.

W.CPSM (20)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING STORAGE MOVE FLAG.

W.CPSTAT (20)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING STATUS BYTE.

W.CPTIME (23)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING CP TIME ACCUMULATED BY JOB.

W.CPTIML (23)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING CP TIME LIMIT IMPOSED ON THE JOB.

W.CPVRND (154)
RELATIVE WORD IN CONTROL POINT AREA WHICH TRANSMITS VISUAL REEL NUMBER TYPED BY OPERATOR TO TAPE LABELING ROUTINE.

W.ECTIME (25)
RELATIVE WORD IN THE CONTROL POINT AREA CONTAINING THE ≠ALLOCATABLE ECS≠ TIME FOR A JOB.

W.EQP (27)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING BITS INDICATING EQUIPMENTS CURRENTLY ASSIGNED TO THIS CONTROL POINT.

W.FSTCC (151)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING FST ENTRY (FNT WORD 2) FOR JOB INPUT FILE. CONTENTS OF THIS WORD DESIGNATE POSITION ON DEVICE AT WHICH NEXT PRU OF CONTROL CARDS MAY BE FOUND.

W.FTYPE (0)
RELATIVE WORD IN FNT ENTRY CONTAINING FILE TYPE FIELD.

W.LBCK (7)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING THE LABEL CHECKSUM.

W.LBDATE (0)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING THE DATE.

W.LBFLAW (10B)
FWA OF THE LABEL FLAW TABLE.

W.LBID (0)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING THE LABEL ID.

W.LBNUM (0)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING THE LABEL NUMBER.

W.LBPFD (2)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING THE PFD POINTER.

W.LBPRIV (3)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING PRIVATE PACK INFORMATION.

W.LBRBTC (2)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING THE RBTC POINTER.

W.LBRC (2)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING THE RECOVERY CATALOG POINTER.

W.LBVID (1)
RELATIVE LOCATION IN THE DEVICE LABEL OF THE WORD CONTAINING THE VISUAL ID.

W.PPIR (0)
RELATIVE WORD IN A PP COMMUNICATION AREA CONTAINING PP INPUT REGISTER.

W.PPMESX (1≤X≤6) (2-7)
RELATIVE WORD IN PP COMMUNICATION AREA CONTAINING SIX WORDS PP MESSAGE BUFFER.

W.PPOR (1)
RELATIVE WORD IN PP COMMUNICATION AREA CONTAINING PP OUTPUT REGISTER.

W.PPTIME (24)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING PP TIME ACCUMULATED BY JOB.

W.RBRLAV (1)
RELATIVE WORD IN RBR HEADER WORD CONTAINING COUNT OF RECORD BLOCKS LOGICALLY AVAILABLE.

W.RBRTPA (0)
RELATIVE WORD IN RBR HEADER CONTAINING EQUIPMENT TYPE AND ALLOCATION STYLE.

W.RBRUNT (0)
RELATIVE WORD IN RBR HEADER CONTAINING UNIT NUMBER (DST ORDINAL)

W.RWPPCW (2)
RELATIVE POSITION OF READP/WRITEP COMMUNICATION WORD IN MESSAGE BUFFER OF PP COMMUNICATION AREA.

W.SSW (26)
RELATIVE WORD IN CONTROL POINT AREA CONTAINING SENSE SWITCH SETTINGS FOR JOB.

W.STCPU (0)
RELATIVE WORD IN STACK REQUEST CONTAINING CONTROL POINT AND UNIT NUMBER (DST ORDINAL) OF REQUEST.

W.STEI (0)
RELATIVE WORD IN STACK REQUEST CONTAINING EMPTY INDICATOR. IF THIS FIELD IS 0, THE ENTRY IS NOT IN USE.

W.STFB (1)
WORD IN STACK REQUEST WHICH CONTAINS FLAG BYTE.

W.STO (0)
WORD IN STACK REQUEST WHICH CONTAINS STACK PROCESSOR ORDER.

W.STPFW (1)
WORD IN STACK REQUEST WHICH CONTAINS NEXT ADDRESS IN PP MEMORY FOR DATA TRANSMISSION. FIELD IS USED IN THIS MANNER ONLY ON CALLS TO R.READP OR R.WRITEP.

W.STPLW (1)
WORD IN STACK REQUEST WHICH CONTAINS LWA + 1 IN PP MEMORY FOR DATA TRANSMISSION. FIELD IS USED IN THIS MANNER ONLY ON CALLS TO R.READP OR R.WRITEP.

W.STPMS (1)
WORD IN STACK REQUEST WHICH CONTAINS PP MESSAGE BUFFER ADDRESS.

W.STPPRU (0)
WORD IN STACK REQUEST WHICH CONTAINS PRU NUMBER AT WHICH TO BEGIN DATA TRANSMISSION IF NO-FNT IS SPECIFIED.

W.STPRBA (0)
WORD IN STACK REQUEST WHICH CONTAINS ADDRESS OF RBT WORD PAIR CONTAINING RECORD BLOCK AT WHICH TO BEGIN DATA TRANSMISSION IF NO-FNT IS SPECIFIED.

W.STPRBN (0)
WORD IN STACK REQUEST WHICH CONTAINS RBT ORDINAL OF RECORD BLOCK AT WHICH TO BEGIN DATA TRANSMISSION IF NO-FNT IS SPECIFIED.

W.STPWC (1)
WORD IN STACK REQUEST WHICH CONTAINS NUMBER OF PP WORDS (BYTES) TO BE TRANSMITTED DURING A READP OR WRITEP REQUEST.

2.14 SCPTEXT MACROS

LDK Macro
Generates LDN or LDC instruction, depending on size of address field. Any symbols in address field must have been previously defined. This macro is recommended for referencing SCPTEXT symbols for CM pointer words.

ADK Macro
Generates ADN or ADC instruction, depending on size of address field. Any symbols in address field must have been previously defined. This macro is recommended for referencing SCPTEXT symbols for control point additives (W.x symbols).

UJK Macro
Generates UJN or LJM instruction, depending on length of jump. In general, the jump must be backward, since symbols used in address field must have been previously defined. Macro is useful for exiting from small subroutines subject to expansion.

BIT Macro
Generates no code; merely defines a symbol in the location field. Value assigned to symbol is a 1-bit mask where the bit is positioned according to the value of address field. Bits are counted from right to left, beginning with zero. Thus, the statement MASK BIT 2 would set MSK equal to 4. Macro is useful for generating 1-bit flag values with the S.x SCPTEXT symbols.

ENM Macro
Generates standard subroutine entry and exit lines. The name of the subroutine is that declared in location field of ENM; the subroutine may be entered by an RJM to that name. If address field of ENM is blank, no exit symbol is defined; otherwise, contents of address field are appended to location symbol to generate subroutine exit symbol. (Typically, address field contains only an X.) An exit from subroutine may then be made by jumping directly to the generated symbol.

PPENTRY Macro
Used as first instruction following ORG in a primary level overlay. PPENTRY generates code to set up low core parameter as follows:

D.PPIRB through D.PPIRB+4	Input register contents
D.CPAD	Control point address
D.RA	Reference address/100B
D.FL	Field length/100B

Address field of the PPENTRY macro should contain: D.PPIRB, D.TO.

LDCA Macro
Load PP A register with absolute 18-bit central address. Relative CM address. Relative CM address is obtained from two consecutive PP low core locations, the first of which is specified in address field of LDCA macro; CM address is assumed to be right justified within these two words. Contents of D.RA are added to CM address. Macro is useful for loading many different CM addresses. Space may be conserved by using a subroutine rather than a macro if the same address is to be loaded three or more times.

CRI Macro
Reads contents of a CM word the address of which is contained in a central memory pointer. Address field of CRI macro contains X, Y, and Z subfields, in that order.

X	6-bit CM pointer word address
Y	First of five PP low core cells which will contain the desired CM word.
Z	Byte within CM pointer word containing 12-bit CM address of desired word.

JOBCARD Macro
SCPTEXT contains a definition of a macro called JOBCARD. The release version is empty, consisting of a macro definition header and a terminator. System characteristics may be altered by insertion (between header and terminator) of one or more cards described below.

If the symbol SCOPE 2.0 is defined within JOBCARD, SCOPE 3.1 is altered to accept only SCOPE 2.0 job cards. The value to which the symbol SCOPE 2.0 is equated is irrelevant.

SCOPE 3.1 may be altered to accept a decimal value on one or more of the job card parameters by inserting a card or cards in the following form:

DECIMAL field

Field is one of the terms EC, CM, T, or P. Currently, all values are assumed to be octal; however, it may be declared specifically that a parameter is to be interpreted as octal by inserting a card in the following form:

OCTAL field

Field is as defined above.

Priority sublevel computation may be tailored by installation as described below.

This procedure combined with the (installation partitioned) 12-bit priority is used to order jobs within priority levels upon entry to the input queue.

The installation partitions the priority byte by specifying a maximum priority level, IP.MPR. The user-supplied priority value from the job card specifies the high order bits (level) of a job's priority. The other job card characteristics may be used as data for priority sublevel computation algorithm. This algorithm is specified by the installation by inserting a set of statements of the form:

WEIGHT field, relation value additive

Field is one of the terms EC, CM, T signifying ECS storage allocation (in 1000_8 word blocks), CM storage allocation (in 100_8 word blocks) and T is Time limit (in 8 second units).

relation is GE (greater than or equal) or LE (less than or equal)

value is a comparison quantity

additive is to be added to the sublevel if the job card field bears the stated relation to value.

The installation may tailor this algorithm to give high priority sublevel to particular classes of jobs (express type jobs). User specified priority may be relegated to its proper role of distinguishing urgent jobs from the great majority of batch jobs which may enter the system at an installation specified standard level (IP.SPR) and be fanned out along a priority spectrum by the input scheduling procedure.

3.0 PERIPHERAL PROCESSOR RESIDENT

INTRODUCTION

In the SCOPE Operating System, the System Display program (DSD) and the Monitor program (MTR) permanently reside in two of the ten peripheral processors, 9 and 0 respectively. The remaining processors, 1-8, form a pool of processors to which MTR may assign tasks as required. These pool processors have no fixed assignments; any processor may be assigned to the execution of any system routine, and it is possible that more than one processor may be executing the same routine at the same time. All ten processors contain a small resident program which handles the communications between pool processor programs and the Monitor and initiates the execution of these programs as directed by MTR.

POOL PROCESSOR STRUCTURE

PP resident is contained in locations 0100 - 0772; locations 75, 76, and 77 contain pointers to the Input Register, the Output Register, and Message Buffer in central memory. When directed to do so by MTR, the resident loads a program into its memory and executes it; since that program remains in that processor only for the period of time required to perform its function, it is called a transient program. Transient programs occupy locations 0773 - 1772, although the first instruction is at location 1000. Transient programs generally load overlays to perform specific tasks. For example, CIO, which is a transient program, calls various overlays depending on the task (read, write, backspace) and the equipment (disk, tape, etc.) specified. Secondary overlays are loaded into memory beginning at location 1773, the first instruction falling at location 2000. Overlays are generally entered via a return jump. Transient programs have names beginning with a letter (CIO, EXU) or the numeral 1 (1BJ, 1LT); overlays have names beginning with a numeral 2 through 7 (2BP, 4LB, 7TP, etc.).

Both transient and overlay programs, as well as the resident program, make extensive use of the low core locations 01-74.

THE RESIDENT

The peripheral processor resident program has two main functions to perform:

All communication between MTR and the transient or overlay programs is handled by the resident.

The resident, when directed by MTR, loads transient programs and initiates the execution of these programs.

Communication between MTR and the resident programs is

carried out through the use of ten communication areas in central memory, one for each processor. Each communication area consists of a one-word Input Register, a one-word Output Register, and a six-word Message Buffer. Pool processors address these areas by means of pointers in locations D.PPIR, D.PPOR, AND D.PPMES1.

MTR assigns a task to a pool processor by placing the request in the processor's Input Register. The name of the program package which is to be loaded and executed appears in the high-order 18 bits of the Input Register. This name consists of three display code characters, such as 1AJ, CIO, etc. The number of the control point to which this package is assigned appears in the low-order three bits of byte 1 of the Input Register. Package parameters, such as the address of arguments required by the package, appear in the low-order 36 bits of the Input Register. The request remains in the Input Register until the task is completed. On completion of a task, the transient program requests MTR to release the processor; MTR then clears the processor's Input Register. The Input Register of a pool processor is thus clear only when the processor is idle.

All communication between the Monitor and the transient and overlay programs is handled by the resident program. MTR performs a variety of functions, each of which is identified by a function code of one or two octal digits.

To transmit a request to MTR, the resident places the request in its Output Register. Byte 0 of the Output Register contains the function code in the low-order bit positions. Bytes 1 - 4 are used for arguments; the number of argument bytes depends on the particular function. Thus, for a Request Channel function (R.RCH=2), the channel number is placed in byte 1. For some functions, the function arguments are placed in the Message Buffer and only the function code appears in the Output Register.

MTR regularly scans the Output Register of each processor to determine if a request is present. When the request has been detected, analyzed, and processed, MTR clears the Output Register. The resident, after placing the request in the Output Register, waits for the Output Register to be cleared before proceeding.

The resident contains a routine called R.MTR which handles the transmission of function requests to MTR. This Process Request routine uses locations D.T0-D.T4 in peripheral processor memory as temporary storage for the request to be written in the Output Register. A peripheral processor program may utilize the routine by placing the arguments for the function in bytes D.T1 through D.T4, setting the A register with the function number, and executing a return jump to R.MTR. Resident

routine will enter the function number in loaction D.T0 and write the contents of locations D.T0-D.T4 in the Output Register. Control will be returned to the requesting program upon MTR's clearing the Output Register.

When a pool processor program completes execution, it exits to location R.IDLE, which is the address of the resident idle loop. In this idle loop, the processor's Input Register is scanned at intervals until a request is found in the Input Register. A delay between successive scans avoids unnecessary memory and read pyramid conflicts. When a request is detected, the resident stores the routine name and the control point number. It then sends function R.PAUSE, pause for Storage Relocation, to MTR and waits for MTR to clear the Output Register before continuing. Should MTR be in the process of relocating the storage assigned to this control point, the Output Register clear will be delayed until relocation is complete. The resident then searches the library directory for the requested routine; if found, the package is read from the resident library into the processor's memory beginning at location 773, and resident turns control over to this routine by jumping to location 1000. If the routine is not found in the directory, the resident enters the message "XXX NOT IN PPLIB" in the dayfile, and requests MTR to abort the job which called the routine. The resident then returns to its idle loop.

3.1 RESIDENT ROUTINES

Several resident routines and words are used by transient and overlay programs. These routines are described below. Values are always subject to change.

R.IDLE (100)

Calling Sequence: LJM R.IDLE

R.IDLE is the idle loop in which PP resident continually scans its input register for something to do.

R.OVLJ (111)

Calling Sequence: Store name of overlay in D.T6, D.T7. LJM R.OVLJ

Go to R.OVLJ to load a new primary overlay and transfer control to it.

R.OVL (124)

Calling Sequence: Load A register Load Address
RJM R.OVL

R.OVL causes an overlay whose name appears in D.T6 and D.T7 (left justified) to be loaded into the PP beginning at the address specified in the A register. R.OVL is used both by PP overlays to load higher level overlays and by PP resident to load the overlay named in the input register. PP resident does not reference the disk directly to load disk resident overlays but makes a call to the stack processor by calling R.READP.

R.EREQS (300)

Calling Sequence: Store L(request) in D.TO

RJM R.EREQS

R.EREQS adds the control point number to the already formatted request and searches the central memory request stack for an empty entry. The monitor function, M.EREQS, is called and PP resident iterates until the monitor accepts the request. If the available flag (bit S.STF+S.STFA of byte C.STFB of the second word of the request) is set, R.EREQS exits to R.IDLE; otherwise it returns control to its caller.

R.WAIT (410)

Calling Sequence: RJM R.WAIT

R.WAIT will cause the PP to idle until byte 0 of the output register is clear.

R.PAUSE (430)

Calling Sequence: RJM R.PAUSE

STD D.RA

R.PAUSE will exit if the PP is attached to control point zero or if the storage move flag is not set. Otherwise, the monitor function, M.PAUSE, will be issued and the PP will pause until monitor has completed the storage move for that control point. In any event, before an exit is made from R.PAUSE, the following will be set:

(D.TO + C.CPST) = control point status

(D.TO + C.CPEF) = control point error flag

(D.TO + C.CPRA) = control points RA (hundreds)

(D.TO + C.CPFL) = control point FL (hundreds)

A-register = control points RA (hundreds)

D.RA and D.FL (if significant) should always be reset after a jump to R.PAUSE.

R.MTR (450)

Calling Sequence: Store function parameters in D.T1 to D.T4

Load function code

RJM R.MTR

R.MTR places the function code in D.TO, writes D.TO through D.T4 to the output register and waits for the output register to clear.

R.PROCES (450)

R.PROCES is identical with R.MTR.

R.READP (R.WRITEP) (460) (470)

Calling Sequence: Load L(request)

RJM R.READP(R.WRITEP)

R.READP (R.WRITEP) computes the PP word count from the first and last word addresses given in the already formatted request and adds the computed word count, the address of the PP message buffer, and the control point number to the request. The request is entered in the stack and data is transmitted via channel directly to (from) PP memory. Upon exit from R.READP (R.WRITEP), the following information will be set:

(D.T3 + C.RWPPWT) = number of PP words transmitted

(D.T3 + C.RWPPLW) = LWA+1 of data transmitted

(D.T3 + C.RWPPST) = upper six bits of status in bits 0-5

(D.T3 + C.RWPPST+1) = lower twelve bits of status

The 18-bit status has the same format and meaning for PP I/O as the status in bits 0-17 of the first FET word for central memory I/O.

R.STBMSK (611)

R.STBMSK is the address within PP resident of a location used by the R.STB routine, q. v.

R.STB (620)

Calling Sequence: Load L(list)

RJM R.STB

where list has the form

L (byte)

L (word 1)

L (word 2)

.

.

.

L (word n)

zero

An entry point to R.STB called R.STBMSK is the address of the mask "anded" with each word in the list before the word is "exclusive ored" with the byte. This mask is initially 7700B and this value should be restored by any routine which substitutes an alternate mask. R.STB is used primarily to substitute channel numbers in driver overlays.

R.CPFL (627)

R.CPFL specifies the location within PP resident which contains the field length/100B of the control point to which the PP is attached. R.CPFL is reset each time the R.PAUSE routine is entered.

R.CPRA (631)

R.CPRA specifies the location within PP resident which contains the reference address/100B of the control point to which the PP is attached. R.CPRA is reset each time the R.PAUSE routine is entered.

R.TFL (634)

```
Calling Sequence:   Load   relative address
                    RJM    R.TFL
```

R.TFL is used to insure that a relative address is within the field length. The 18-bit address is added to the control point reference address (RA) and compared with the field length. If the address is out of range, R.TFL will exit with a negative A register; if the address is legal, the A register will contain the absolute CM address (RA + relative address) upon exit. The control point RA and FL are kept locally within PP resident at R.CPRA and R.CPFL, respectively; these locations are reset when an entry to R.PAUSE is made.

R.DFM (650)

```
Calling sequence:   LOAD   L(message)+flag bits
                    RJM    R.DFM
```

R.DFM will cause a message to be written from PP memory to the dayfile and/or the console. The flag bits are contained in the high-order 6-bits of the A register upon entry to R.DFM and are used to determine the destinations of the message. Possible values of the flag bits are described below; one or more bits may be on. All are optional.

1 = Dayfile only (B Display)

2 = Control Point 0 (System) message

4 = System Dayfile (No A Display)

10B = Collation (Accounting) Flag. If set then a $ will be placed in the 20th character of messages that are sent to the system dayfile (not set by any CDC routine. Used by installations).

20B = C. E. Error File. Note: C. E. Error File entries are unique. See section 6.7 for proper formats and calling procedures.

R.RCH (704)

Calling Sequence:

Load channel number

RJM R.RCH

The channel numbers contained in the A register will be stored in byte D.T1, monitor function M.RCH inserted in D.T0, and D.T0-D.T4 written to the output register for that PP. Channels will be assigned by MTR on the following priority basis:

D.T0	D.T1	D.T2	D.T3	D.T4
	2 | 1	4 | 3		

If alternate channels are specified MTR will stop looking for alternate channels upon sensing 6 bits of zero. Thus, if one alternate channel is desired, the programmer must clear D.T2 before entering R.RCH so the search will be terminated at that point. The procedure for requesting channel 12 with alternate channel 13 would be:

```
LDN   0
STD   D.T2
LDC   1312B
RJM   R.RCH
```

Monitor will stop looking for alternate channels after four channels have been investigated.

When R.RCH is used, D.T4 is automatically set nonzero; in this case, the function is not considered complete (i.e., output register is not cleared) until a channel can be assigned. When complete, byte 0 of the output register is cleared.

R.DCH (714)

Calling Sequence:

LOAD channel number

RJM R.DCH

R.DCH will cause the specified channel to be dropped.

Four categories of direct cells are used by the PP resident subroutines. Under each category are listed all routines which, together with all subroutines they may call, use only cells in that category.

Cells 1 - 17 (D.Z1 - D.T7)

R.OVL

Cells 10 - 17 (D.T0 - D.T7)

R.READP
R.WRITEP
R.EREQS
R.DFM

Cells 10 -14 (D.T0 - D.T4)

R.RCH
R.DCH
R.PAUSE
R.PROCES
R.WAIT

Cells 10, 12 (D.T0, D.T2)

R.STB

No direct cells

R.TFL

All subroutines except R.TFL destroy cell 0.

CHAPTER 4 - TABLE OF CONTENTS

4.0 Dead Start 4-0

4.1 Introduction 4-1

4.2 General Flow 4-3

4.3 CM Allocation 4-11

4.4 CONTROL 4-13

- 4.4.1 CED and D 4-15
- 4.4.2 EST - Process Equipment Changes 4-18
- 4.4.3 P 4-19.1
- 4.4.4 IRP 4-20

4.5 IRCP 4-29

- 4.5.1 Preliminary IRCP Tasks 4-29
- 4.5.2 Preloading 4-29
- 4.5.3 Loading 4-33
- 4.5.4 Recovery 4-39
- 4.5.5 Label and Permanent File Processing 4-40
- 4.5.6 IRCP General Subroutines 4-47

4.6 Dead Start Common Decks 4-50

4.7 Hardware Aspects of Dead Start 4-50

4.8 Operator Communication Package (OPCOM) 4-59

4.9 Dead Start Debugging Aids 4-65

4.1.0 Introduction

The Dead Start package, although formally a part of the SCOPE Operating System, performs functions beyond the control of SCOPE, the most important of these being the loading and activating of SCOPE itself. Dead Start and the system have no programs in common (with the trivial exception of some disk driver code), and the running of one never overlaps with the running of the other. The functions of Dead Start fall into two categories: 1) activation of the SCOPE system; and 2) certain stand-alone utilities, such as post-mortem dump, etc. Following system activation, no further Dead Start functions are performed until Dead Start is reactivated by toggling the Dead Start switch. Following completion of a utility function, another Dead Start function must be performed by toggling the Dead Start switch.

4.1.1 Dead Start Functions

Dead Start and the operator communicate by means of the 6602/6612 Console Display. Dead Start displays information about, and the operator can control at appropriate places, the dead start processing. The basic control is selecting the function to be performed.

4.1.1.1 System Activation

Activating the system consists of either loading a fresh system or restoring a previously running system, passing certain information to that system, and then turning over control of the machine to it. The passed information is in the form of CMR tables, Entry Point and Program Name Tables, and RBT's in CM, and the Permanent File tables on some mass storage device. The types of system activation are as follows.

1. Initial Dead Start
   Write device labels on all RMS devices, set up an empty PF Directory, load the system from tape, and turn over control.

2. Normal Dead Start
   Verify existent labels on all RMS devices, restore the existing PFD and all active permanent files, load the system from tape, and turn over control.

3. Recovery
   Verify existent labels on all RMS devices, restore the existing PFD and all active permanent files, recover non-permanent files, restore the system, and

turn over control. The system can be restored in one of four ways:

A. Load a fresh system from the tape.
B. Restore the original system from the disk.
C. Restore the system from the disk retaining all EDITLIB's but the last completed one.
D. Restore the system from the disk retaining all completed EDITLIB's.

Through recovery, the operator can also bypass the recovery of user files and simply perform a dead start from the disk by treating the system as described in B above.

4.1.1.2 Utility Functions

1. Dead Start Dump
   The operator can dump selected portions of Central Memory, PP memory, or ECS and write the results to either a printer or a tape.

2. 6603II Pre-addressing

4.1.2 System Tape

The SCOPE 3 system tape as released is composed of several "special" records at the beginning, a record containing the Entry Point Table, a record containing the Program Name Table, and then the library records. The "special" records consist of 13 Dead Start programs, MTR, DSD, and from 1-8 copies of CMR. The system tape is ordered as follows:

1. CEA
2. CED
3. D
4. N copies of CMR

N + 4. EST
N + 5. IRP
N + 6. SCP
N + 7. SCQ
N + 8. SCR
N + 9. SCS
N + 10. SCT
N + 11. P
N + 12. STL
N + 13. IRCP
N + 14. MTR
N + 15. DSD
N + 16. EPT
N + 17. PNT
N + 18. 1st library program
.
.

The number of special records is variable and can be changed at will. All records following CMR are located by name, hence additional programs can easily be inserted. When doing so, the relative order of the existing routines should remain the same. (Note: EDITLIB has been modified such that the LIST function searches for DSD rather than skipping a fixed number of records. Consequently, DSD must immediately precede the Entry Point Table.)

The Dead Start package consists of four decks: CEA, CONTROL, STL, and IRCP. CONTROL in turn contains CED, D, EST, IRP, SCP,..., SCT, and P as segments. In addition the following common decks are used: DSLCOM (basic Dead Start parameters), DSMAC (operator communication macros), RMSA (6603-I driver code), RMSB (6638 driver code), RMSC (6603-II driver code), RMSD (865 driver code), RMSP (854 driver code. ECSCOM is called if the BNL ECS code is assembled.

4.2.0 General Flow

4.2.1 CEA

When the dead-start button is pressed, CEA is read from the tape and given control. CEA is a very small program whose only function is to send the contents of PP0 to central memory. The only reason for CEA existing is so that the first record to be read from the tape will be quite short, and hence not overwrite much of PP0. CEA then uses the code sent to PP0 via the dead-start panel to read the next tape record. This record must be the program CED.

4.2.2 CED

CED is a relatively large routine which contains the following:

1. A routine which determines what type of dead-start is to be performed.
2. Routines to perform preliminary dead-start functions, such as freeing the PP's from their respective channels.
3. Routines which will be resident in all PP's that are involved in the dead-start process. These include a display routine and tape I/O routines

4.2.3 Dead-start Dump

The first decision for CED is whether or not the dead-start

being performed is for a dead-start dump. Thus, the first message to appear on the display asks if a dump is desired. If the operator asks for dump, CED reads another tape record directly into PP0. This is expected to be the dead-start dump program. Control is given directly to dump. At this point nothing has been overwritten except PP0, six bytes at the beginning of the other nine PP's and the area in central memory where PP0 was saved by CED. A switch on the deadstart panel can be used to suppress the saving of PP0 if it is desired to get an accurate dump of this area in central memory.

4.2.4 CMR and Dead-start Options

If dump was not selected, CED is free to make use of central memory and the other PP's. The first task at this point is to initialize CMR. The contents of the CMR present at the time of dead-start are saved in a remote area in central memory. This will be used if a recovery-type dead-start is being performed. After saving the old CMR, CED loads CMR from the system tape.

Once CMR is loaded, the operator is presented with a display which shows all of the available options. Any of these may be changed, and all assumed values are shown. At this point, the operator may also make changes in the equipment configuration.

4.2.5 Mass-Storage Drivers

When the operator has made all desired entries, CED first loads drivers for each type of mass-storage device. These drivers are read from the tape and placed in an area in central memory where they may be found by the routines that need them later.

4.2.6 Types of Dead-Start

At this point, what CED does depends on what general type of dead-start is being performed. There are two types: 1) Utility, and 2) system activation. The utility type dead-start involves any function which does not involve the IRCP - IRP interface (described below). Such routines perform only limited tasks and are unable to initiate the operating system.

4.2.7 Pre-addressing

There is currently only one dead-start utility function. It is the pre-address routine for the 6603-II. If this type of dead-start is selected, CED reads the routine named P from the system tape and sends it to PP1 where it then executes. P reads the 6603-II driver from central memory (placed there by CED) and does all disk I/O within PP1. The only thing not actually done by P is the driving of the display. PP0 acts as a display driver for any utility routines. When pre-addressing is completed, PP1 simply hangs up, and another dead-start must be performed.

4.2.8 System Activation

4.2.8.1 Setup

The IRCP-IRP type dead-start is the much more important of the two types, and will require much more discussion. Its purpose is to initiate the execution of the SCOPE operating system in one of several ways. However, its first responsibility is that of recovering permanent information on any of the mass-storage devices known to be within the configuration.

As described above, the mass-storage drivers have been loaded into central memory. In addition, the routine IRP is sent to PP1 and given control. IRP consists of the code necessary to do tape and mass-storage operations. The code for mass-storage I/O (commonly called RMS control) is sent over to PP2 and allowed to begin execution. Also, after loading IRP, CED reads IRCP from the tape and loads it into central memory. IRCP is a central processor program which does most of the work remaining to be done by dead-start. CED starts up IRCP by an exchange jump and then begins to function entirely as a display driver.

Thus, at the time IRCP begins to execute, the following conditions exist:

1. IRCP is executing in the central processor.
2. PP0 is executing as a display driver, and may display any messages requested by the central processor or any PP's.
3. PP1 contains the tape I/O routine. It is waiting for a request to process.
4. PP2 contains the RMS control routine. It is waiting for a request to process. It will make use of the drivers that were stored in central memory.

4.2.8.2 Functions

The functions performed by IRCP depend on the type of dead-start that was selected by the operator. There are three types of dead-start processed by IRCP. The following summarizes the procedures followed for each of the three types:

1. Initialization - type
   a) Label writing
   b) Label checking
   c) Preloading
   d) Loading

2. Normal - type
   a) Label checking
   b) Preloading
   c) Loading

3. Recovery - type
   a) Label checking
   b) Recovery
   c) Preloading (in certain cases)
   d) Loading

Each of the above processes is described briefly in the following section. A more detailed description appears in the IRCP section.

4.2.8.2.1 Label Checking

Label checking involves accessing every mass-storage device except those turned off and searching for a label. This is done before pre-loading, recovery, or loading because it involves designating that certain record blocks are not to be written on by the dead-start process. The label serves two purposes:

1. When the device is encountered which contains the Permanent File Directory (PFD) and the RBT catalogue (RBTC), permanent file information is set up in CMR. This includes reserving all record blocks containing permanent information.

2. For all devices, a flaw table is kept in the label. When the label is read, all flaw bits are placed in the Record Block Reservation (RBR) table. The flaw table is never modified other than at dead-start time.

If a device is encountered with no label, the operator must designate that a label is to be written on the device. The label is then written before any other processing takes place.

4.2.8.2.2 Label Writing

Labels are normally written on every mass-storage device except those turned off during an initialization-type dead-start, and also when so instructed by the operator after a label could not be found on a device. The operator is given complete control over the label writing process. Just prior to writing a label for a device, a display is generated which shows the following:

1) Device information (type, channel, etc.)
2) Whether or not the device already has a label. If a label was found, the remaining information is obtained from the label.
3) Date
4) Visual Identification
5) Whether or not this is the primary Permanent File device (device containing PFD and RBTC).
6) Contents of flaw table.

The operator may change any of items 3-6 above as he pleases. To free the operator of numerous type-ins at times he knows they are not necessary, he may indicate that all remaining labels be written without any additional type-ins. It is also possible to skip writing of the label for one or all remaining devices, except in this case, the operator will be informed if a device is found without a label.

4.2.8.2.3 Preloading

The process of copying the contents of the system tape to one or more mass-storage devices is referred to as preloading. The information which is written during this process becomes the system file (logical file SYSTEM). The following should be noted:

1) Tape input is performed by issuing requests to the tape I/O routine which is executing in PP1. Each request causes one tape record (PRU) to be read.

2) The input data is formatted suitable for output to the mass-storage device. This involves moving the data to one of the output buffers, and, at the same time, inserting word count and checksum bytes at the start of each logical PRU to be output.

3) Output to the mass-storage device is performed by the RMS control routine in PP2. IRCP issues one request to RMS control each time a record block is to be written. The write-with-RBT order code (04) is used, and, as a result, the entire RBT chain for the system file is constructed by RMS control.

4) More than one system device is used only if the system file will not fit entirely on the first system device. As many as five devices will be used, if necessary.
5) Since preloading is the most time-consuming of the dead-start processes, double buffering is used for both the tape and RMS I/O. All three of the operations described above proceed concurrently. The system tape will read non-stop while pre-loading to any type of mass-storage device whose driver produces a transfer rate greater than that of the tape. This is currently the case for all devices except the 854.

4.2.8.2.4 System Loading

The dead-start loader portion of IRCP is called either at the end of the preloading or recovery processes. It involves reading SYSTEM from the system device(s), placing the CM resident library in CM, and constructing the appropriate tables to allow the system to assume control. More specifically, the following functions are performed:

1) STL, MTR, DSD are saved in CM at STLBUF, MTRBUF, and DSDBUF for later use.
2) The Entry Point Table and Program Name Table are placed in CM immediately following CMR.
3) Each program in the system library is processed as follows:
   a. Its checksum is validated.
   b. Its residence is determined and if
      1. DS resident, its disk address is written in the PNT
      2. CM resident, it is copied to the next available location in the CM resident library and its CM address is written in the PNT.

   (Note that only the two residence types are presently allowed.)
4) Miscellaneous fields in CMR which have not yet been set up are set up at this time. Finally, signals are sent to accomplish the following:
   a. Stop PP0 from driving the display.
   b. Stop PP1 from looking for tape requests. PP1 then loads and executes STL.
   c. Stop PP2 from looking for RMS requests.
5) A stop instruction marks the end of the loading process and, in fact, of all IRCP functions.

4.2.8.2.5 Recovery

The purpose of dead-start recovery is to restore the whereabouts of certain files that existed in the system at the time of dead-start. The basic steps are as follows:

1) At the beginning of the dead-start process, the contents of CMR as it existed at dead-start time are saved starting at TABLESAV. The RBT's are saved at RBTSAVE.
2) The fresh CMR is loaded from the system tape, and an empty RBT chain is initialized.
3) The recovery routine gets the information about previously existing files from the FNT and RBT's saved in (1) above. Note that certain tables must be intact at the time of dead-start in order for recovery to be successful. These include the FNT, all RBT's, and the system dayfile buffer. These tables are restored from the old CMR for every recovery, and it is not necessary to perform two dead-starts to accomplish this as it was with the previous version of dead-start.
4) System loading is performed so as to give control to the operating system.

4.2.9 STL

STL is the last dead-start routine to be executed before the SCOPE operating system gets control. It is loaded into PP1 as soon as IRCP has completed the dead-start loading process. It contains PP resident plus the code necessary to initialize the remaining PP's. The following takes place:

1) All PP's are currently waiting for their respective numbers to be stored in CM location zero. PP0 is waiting for 12B, however, since CM location zero is normally kept equal to zero. When a PP detects this, it clears CM location zero, activates channel zero, and inputs over channel zero to location zero. STL now sends PP resident to PP2 through PP8. To accomplish this, it does the following for each PP:
   a. The direct cells D.PPIR, D.PPOR, and D.PPMES1 are set with the correct addresses of the PP's input register, output register, and message buffer, respectively.
   b. The PP number is stored in CM location zero.
   c. STL waits until the PP picks up the signal, that is, when CM location zero is reset to zero.
   d. STL outputs one byte equal to 77B over channel zero. Since the PP will read this to its location zero, this will cause the PP to begin execution at location 100B (R.IDLE) when channel zero is disconnected.
   e. STL outputs 777B bytes starting from location one. This includes direct cells 1-77B and all of PP resident which resides at 100B through 777B. The PP will read this to the same locations in its memory.

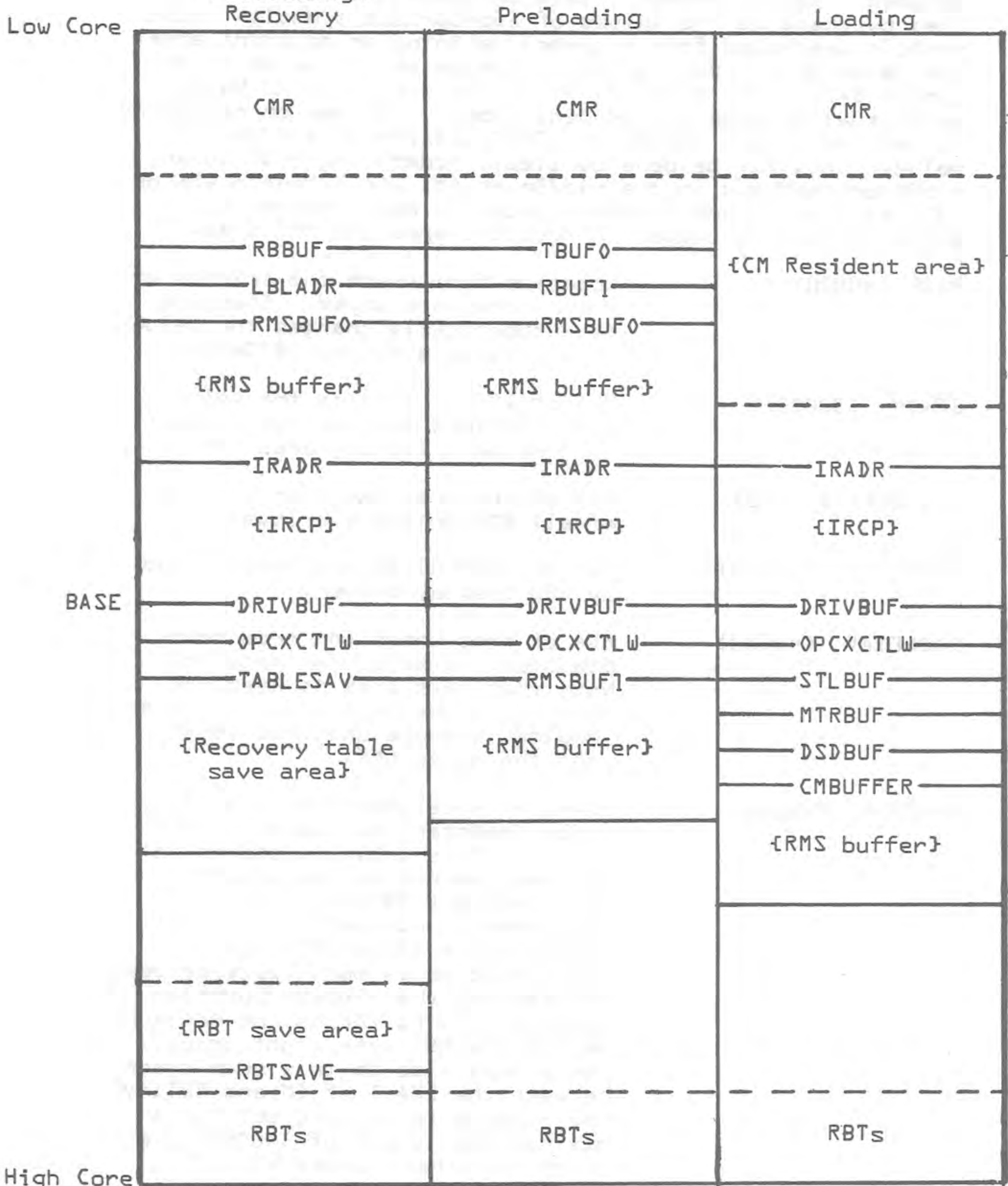
f. STL disconnects channel zero. This causes the PP to begin execution.

2) MTR and DSD are sent to PP0 and PP9, respectively. This is done by signalling to the PP as described above, except only a very small program (seven bytes in length) is sent across channel zero. This small program reads into the PP from MTRBUF or DSDBUF to accomplish the loading and initiation of MTR or DSD, respectively.

3) All PP's are now completely set up, including PP1 (STL), which has had PP resident in place in its own memory since it was loaded. STL thus exits by simply jumping to R.IDLE.

4.3.0 DEADSTART CENTRAL MEMORY ALLOCATION

The following shows how central memory is allocated during the three phases of the deadstart process: (1) Label and Permanent File Processing and Recovery, (2) Preloading, and (3) Loading.



All of the symbols described in this section are defined in the common deck DSLCOM. Their values at the time of the 3.1.6 release are shown. Note that a 32K memory is assumed. Central memory usage by deadstart may be modified by changing the values of appropriate symbols. Most symbols are keyed from a symbol defining an adjacent area. The value of all such symbols depends on the value of the symbol BASE. For example, if a 131K system is to have an unusually large CM resident library, it may be necessary to set the origin address of IRCP (IRADR) to a higher value. This may be done by simply redefining BASE to any arbitrary address in the middle of CM. In either a 65K or 131K machine, there is ample space to make changes to allow for both a bigger CM resident area and RBT area.

BASE (40000)	Location from which the origins of other areas are keyed. Changing it automatically changes the values of all symbols except RBTSAVE.
IRADR (30000)	FWA of IRCP. This is the value which defines the maximum size of the resident library area.
DRIVBUF (40000)	FWA of the area used for storage of all RMS driver overlays.
OPCXCTLW (42000)	FWA of central memory buffer used by PP0 display driver.
TABLESAV (42500)	FWA of area where CMR from prior deadstart is saved for recovery. Note that this area is used for other purposes beginning with preloading, because only recovery uses the saved CMR.
RBTSAVE (64000)	LWA+1 of area where RBT area from prior deadstart is saved for recovery. This table is stored in the same manner as the actual RBT's so that RBTSAVE-1 in the save area is equivalent to LWA+1 of CM in the actual RBT area. There must be as much space between RBTSAVE and the highest location used for saving CMR as the maximum length the RBT area might occupy. There must also be this much space between the LWA+1 of CM and RBTSAVE The release values are set for a maximum RBT length of 14000B words, which is rarely exceeded. The theoretical maximum, however, is 20000B words.

RMSBUF0 (17700)	FWA of an RMS (rotating mass storage) buffer. This buffer is used for all reading and writing of record blocks during label and permanent file processing. It is also used during preloading along with RMSBUF1. However, it is not used during loading because it is intended that the resident library will expand into this area.
RMSBUF1 (42500)	FWA of an RMS buffer. It is used only during preloading in conjunction with RMSBUF0
TBUF1 (16700)	FWA of tape buffer. It is used in conjunction with TBUF0 during preloading. As with RMSBUF0, this space must be relinquished during loading for the resident library.
TBUF0 (15700)	FWA of tape buffer used in conjunction with TBUF1 during preloading.
LBLADR (16701)	FWA of area used for reading and writing of labels during label and permanent file processing.
RBBUF (15701)	FWA of area used for building an RB chain during permanent file processing.
STLBUF (42500)	FWA of area where STL is saved during deadstart loading.
MTRBUF (44200)	FWA of area where MTR is saved during deadstart loading.
DSDBUF (45700)	FWA of area where DSD is saved during deadstart loading.
CMBUFFER (47400)	FWA of RMS buffer used during deadstart loading.

4.4.0 CONTROL

CONTROL is the name of the UPDATE deck which contains in several segments most of the dead-start PP code. The segments are as follows:

1. CED

This routine takes the computer from the dead-start condition, determines from operator entries what type of dead-start is to be performed, sets up CMR (except when dead-start dump is being used), and calls in IRP and IRCP to carry on the dead-start process.

In addition, CED contains several resident routines which reside in the low part of PP0 memory. These are sent over to PP1 along with IRP. These routines include the display drivers which are used only in PP0 and the PP OPCOM routine and tape drivers, both of which are first used by CED, but then only by IRP once it has been sent to PP1.

2. D

This is the dead-start dump routine. When requested by the operator, CED loads it directly into PP0 and executes it.

3. EST

This routine processes changes to the equipment configuration. It executes in PP1 and is used only if the operator desires to make equipment changes.

4. IRP

This routine contains the tape and RMS control routines. The entire IRP routine is sent to PP1 by CED. The first thing IRP does is send the RMS portion to PP2. During the remainder of the dead-start process, PP1 processes tape requests made by IRCP. The tape driver subroutines in low core are used for the actual I/O. PP2 processes all RMS requests from IRCP by using the drivers described below.

5. 5CX Drivers

These are the drivers for the various types of RMS devices. Before CED starts up IRP, it reads these drivers from the tape and moves them to a save area in central memory. When RMS control processes a request, it fetches the appropriate driver from central memory, if necessary, and uses it to do the actual I/O.

There are currently five such routines, as follows:

a)	5CP:	6603	disk driver
b)	5CQ:	6638	disk driver
c)	5CT:	6603II	disk driver
d)	5CR:	865	drum driver
e)	5CS:	854	disk pack driver

6. P

   This routine is used for pre-addressing the 6603II disk. When this option is selected by the operator, CED sends P to PP1 and gives it control.

4.4.1 CED and D

The first thing CED does is issue a tape read so as to read in the second PRU of CED. Then the first dead-start message is displayed. It asks the operator if a dump is desired. If so, CED loads the dump segment (D) and transfers control to it. Up to this point, nothing has been changed in CMR, except for the PP0 save area and nine words that are used for controlling the PP's 1-9 up to this time. These nine words are restored before dumping, however.

If dump was not selected, the old CMR is saved, and a larger idle loop is sent to each PP. This loop causes each PP to look for its PP number in location zero of central memory (see section 4.7.6). CMR is now read from the tape. The tape may contain up to eight different copies of CMR. This allows a tape to be configured for more than one computer. The setting of a certain 3-bit field on the Dead-Start Panel determines which CMR will be loaded. This field is in bits 6-8 of the tape connect code (see figures 4.7.2 and 4.7.3). The value of this field is the number of CMR's that are skipped; that is, with this field equal to zero, the first CMR is used. CED assumes that all copies of CMR follow immediately after the dump (D) segment. It also assumes that there are at least as many copies of CMR present as the setting on the panel indicates.

CED now calls the subroutine MTXCTL to display the various options and to allow the operator to select those which he desires. MTXCTL returns to CED with the parameter word CEDARGS set up for use by the remainder of deadstart. Its format is as follows:

```
byte 0 - unused

byte 1 - 0 if ECS up
         1 if ECS down

byte 2 - unused

byte 3 - X00  if system file is not to be recovered
         X01  if system is to be recovered as before all EDITLIBS.
         X10  if system file is to be recovered as before the last
              EDITLIB.
         X11  if system file is to be recovered with all EDITLIBS.
         0XX  if other files are to be recovered
         1XX  if other files are not to be recovered

byte 4 - 0 if initialization
         1 if normal deadstart
         2 if recovery
         3 if the device debug routine is to be executed
           (See section 4.9.3)
```

During the execution of MTXCTL, the operator may indicate whether or not he wants to make any changes to the equipment configuration. When MTXCTL returns to CED, a flag is set to so indicate. If equipment changes are desired, CED now searches the tape for the segment EST and sends it to PP1. It is assumed that EST follows immediately after the last copy of CMR. The EST routine is described in section 4.4.2. While EST is executing, CED does nothing more than drive the display. EST signals that it is finished by storing a 12B in location zero of central memory.

CED next proceeds to set things up for the next routines to take over control. Starting at location CEDARGS+1 in central memory, a list of up to five EST ordinals are stored, one per CM word. If a pre-addressing deadstart was selected, the EST is searched of 6603-II entries. Otherwise, ordinals for system devices are stored. Now CED begins reading the tape and fetching all necessary routines. The routine IRP (P if pre-addressing) is sent to PP1 and allowed to begin execution. The RMS drivers are stored in central memory, and their names and locations are entered into a directory (DIRECT). Any other routines are discarded until IRCP is encountered, and, at which time, it is assumed that all required routines have been found.

At this point, unless a pre-addressing type deadstart was selected, an exchange jump (EXN) starts IRCP. All of the setting-up functions of CED are now completed. PP0 now becomes the display driver for the rest of the deadstart routines, and IRCP (or P) takes over control of the deadstart process.

The following is a list of the main routines in the resident portion of CED. They are the routines that execute in other PP's. The resident portion ends at the address ORGADRS.

The following are the tape routines:

READTPE	reads the tape into PP memory.
COPYLOG	reads the tape into Central Memory.
READREC	connects the tape unit.
FUNC	issues a function to the tape channel.
TAPSTAT	checks the status of the operation.
CHECK81	checks the status of the 6681.
STAT	reads the tape and converter status.
D81TPE	deselects the 6681 on the tape channel.

The following are utility routines:

INCHNO	inserts channel numbers into specified instructions.
SIGPP	signals a specified PP to come out of its idle loop.
MODIFY	modifies instructions for RMS driver overlays.
ERRINF	processes error information for RMS driver overlays.
RELEASE	disconnects the display channel.

The following are the various components of the OPCOM package. This is the last portion of the resident area:

OPCOM	contains the control portion of the package. It sets up the calling sequence for use by the other parts of the package to create a display driver and processes the resultant input.
MSGTXT	processes messages in a MSGLST list.
QSTEXT	processes messages in a QSTLST list.
OPTEXT	processes messages in an OPTLST list.
CNVTOTB	converts octal display code to binary.
CHKCHR	determines if a character is octal and returns its value if it is.
CNVTOTBA	converts a series of octal digits to binary.
CNVTBTO	converts binary to display code octal.
COORDFIX	converts the line/character co-ordinate to a physical console co-ordinate.
PUTBYTE	transmits a display byte to the display buffer in PP zero or CM.

The following routines are resident in PP0 throughout the deadstart process:

IKSD	is the internal display driver controlling routine for a PP zero OPCOM call.
BLNKOUT	clears the keyboard entry buffer.
DSPINIT	does initialization before the display is started.
ACCPIN	calls DSPDRIV to display the keyboard buffer, calls TKPCTL to read the keyboard, and checks for a carriage return.
TYPCTL	is the keyboard monitor.
DSPDRIV	is the CRT driver.
CEXTRCT	gets a specified character from the keyboard buffer.
CINSERT	puts a character into the keyboard buffer.
CONTROL	is the main loop which controls the first part of deadstart (see narrative of program flow for more details).
KSDA	is the display controlling routine of the display driver for external (to PP zero) calls for displays.
SENDPRG	sends a program read into PP zero memory to PP1 with the resident portion of CED.

The following routines are used only while CED is in control. These routines begin at the address LOADORG. This is also where PP routines to be stored in other areas are brought into PP0 memory.

BEGINPT	starts the PP's inputting on their channels again after they have been idled down reading a CM word repeatedly.
SENLOOP	sends a large (fourteen word) idle loop to each PP.
MTXCTL	displays the deadstart options and accepts operator input to select the desired options.
SAVECMR	moves CMR to a safe place in case it is needed for recovery.
FRECHAN	sends an idle loop to each PP over the channel which it is reading to quiet the channel. At the end of the loop, all channels are free and each PP is reading a word in CM.
PKDATA	saves data from the D/S panel.

The dump segment D consists of the following routines:

OBEY	is the main loop which asks the operator what he wants to do and transfers to the appropriate routine to do it.
OBEYP	is the routine which processes requests to dump PP's.
CONVERT	is the routine to convert the address of CM or ECS which the operator wants to dump. It is used by OBEYC and OBEYX.
OBEYC	is the routine which processes requests to dump central memory.
OBEYX	is the routine which processes requests to dump Extended Core Storage.
WRICP	writes the ECS read program to CM.
XJMP	writes the exchange jump package.
FIXCHAN	sets up each PP with an idle loop in preparation for dumping.
CNV6	converts six bits to octal display or printer code.
CNV12	converts two six bit quantities to octal.
CNV12A	calls CNV12 and accumulates a bit sum.
TPRINT1	keeps track of the number of zero lines and turns off the print after two successive zero lines until a non-zero line is to be printed.
BLANKIT	sets the dump output line to blanks.
EJECT	causes a page eject on printing. If this routine is called and output is to tape, a "1" is put in the first character position of the next line to be written on the tape.
LABEL	Processes requests for labels to be written on tape.
EOF	writes an end of file on the tape.
WRT	handles the physical I/O for tape.
BKP	backspaces the tape one record.
OUTPUT	makes the switch between tape or printer for dump output.
DWRITE	handles the buffering of the dump output to tape.
PRNTCM	handles the conversion of four CM words to form a print line.
PPZERO	gets the bytes of the contents of PP zero which was stored in CM at the beginning of deadstart.
WAITR	waits for the printer to be ready for I/O.
PRINT	prints on the printer.
MACSIZE	determines the size of central memory.
SETPR	sets the printer channel in printer I/O instructions.
SENDPPS	sends the dump loop to each PP.
PREOBEY	does the initial setup for dump.
OBEYT	handles the change from printer to tape for dump output.
OBEYE	handles the change from the assumed printer channel and equipment numbers.

4.4.2 EST - Process Equipment Changes

EST is the name of a segment of the deadstart deck CONTROL. It is used only if the operator desires to make one or more changes to the equipment configuration as indicated on the system tape. CED reads it from the system tape, sends it to PP1, and starts it executing. Until EST is done, CED does nothing more than drive the display.

The contents of the equipment status table (EST) are formatted and displayed on the left screen. The right screen provides a summary of the allowable keyboard entries. When an entry has been entered, the appropriate changes are made to the EST and, if necessary, to other tables. The display of the EST is then formed again and the operator makes another entry. This sequence is repeated until the operator indicates that he has made all desired changes. He does this by entering a lone carriage return. EST then notifies CED of its completion.

A detailed description of all type-ins is not included here: the SCOPE Operator's Guide should be consulted for this.

The following describes how tables are modified:

1. Only the EST is modified if the change deals entirely with non-allocatable devices, or if an allocatable entry is being changed only to the extent of modifying status bits.

2. If a change involves removing an allocatable device entry, all RBR's corresponding to that EST entry are cleared. DST entries are not cleared. This type of change results from a request to clear out an EST entry or from a request to replace an allocatable device entry with an entry for a different device.

3. When a change requires adding an allocatable device to the EST, an RBR is built. In order that this can be done, there must be sufficient space in the RBR area for one RBR. If there is not space, the type-in is treated as an error. The left screen display always shows how many RBR's can be formed in the available space. An RBR can always be inserted if the operator requests that the new entry be inserted in the EST so as to replace another allocatable entry, since the tables for the entry being removed are deleted as described in 2 above.

   A DST entry is added unless there is already one present for the same controller. That is, a similar DST will be identical with respect to driver name, channel number, and equipment number.

The following equipment tables are used by EST. They are generated by calling macros which are defined in the common deck DSLCOM. They are used for building EST, DST, and RBR tables and for identifying devices as 3000-6000 type and allocatable - non-allocatable.

DEVNAMES	contains device mnemonics for allocatable devices.
DRIVERS	contains the driver names of allocatable devices.
DEVTYPES	contains the 6-bit device type codes for allocatable devices.
DEVSIZE	contains the number of record blocks on each allocatable device.
DEVALLOC	contains the allocation style of each allocatable device.
CLASS	contains the hardware mnemonic for all types of devices known to the system and a code as follows:

0 - 3000 allocatable
1 - 6000 allocatable
2 - 3000 non-allocatable
3 - 6000 non-allocatable

If a hardware mnemonic is encountered which is not in this table, it is assumed to be of type 3000 non-allocatable.

4.4.3 P - Deadstart Pre-address 6603-II (Option 10124)

Each 6603-II sector consists of 504B bytes; the last two bytes are reserved for addressing. The P routine pre-addresses the disk by clearing the entire surface and writing an RB number and physical sector address in the address bytes for use during repositioning.

When the pre-address option is selected, CED loads the P routine into PP1 where it is immediately executed. During execution, operator communication is maintained through CED display drivers in PP0 and operator input returns via central memory.

This routine is a stand-alone utility dead-start function. When it is completed, another dead-start must be performed.

4.4.4 IRP

IRP is loaded by CED to drive tapes and rotating mass storage (RMS) for deadstarting.

IRP is loaded by CED when the operator selects deadstart options I (initialize), N (normal) and R (recovery); it is not loaded when options D (dump), and P (pre-address 6603-II) are selected.

IRP consists of three logical parts:

1. Tape drivers and the display format routine from CED.
2. The tape I/O control routine.
3. The RMS (Rotating Mass Storage) I/O control routine.

CED loads IRP into PP1 and transfers control to it. IRP copies the tape drivers and display format routine, and the RMS I/O control routine, along with common utility routines into PP2. PP1 and PP2 can now work independently, fulfilling requests put into their respective message buffers as described below.

4.4.4.1 IRP - Tape Routine

Deadstart system tape READ, REWIND and UNLOAD functions are provided by the IRP tape routine in PP1.

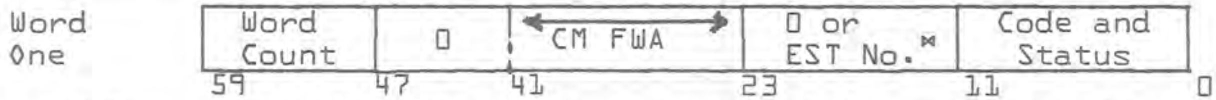
Status Communication

During deadstart, the PP communication area is used differently than during normal operations. The last byte of the PP1 input register is used for status communication during deadstart. If this byte is zero, a request is waiting in the message buffer; if non-zero, there is no request waiting.

IRCP makes a request of IRP by placing the request into word one of the message buffer and setting the last byte of the PP1 input register to zero. IRP senses the "request-waiting" status, executes the request, and resets the input register byte to non-zero (awaiting request) status.

The activity of the IRP tape routine is finally terminated when the input register byte is set to 77B. At this time, the deadstart subroutine STL will be loaded and control transferred to it.

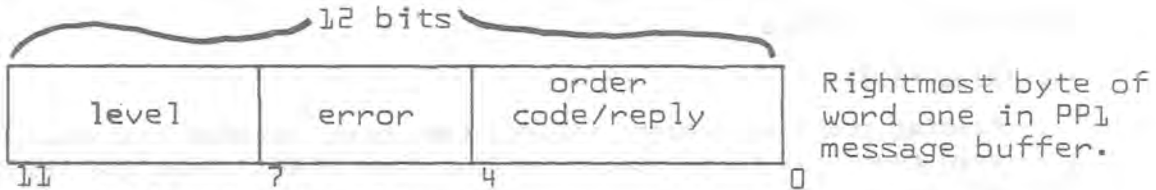
Format of Tape Request in PP1 Message Buffer



* EST No. is to be entered when a request for a tape unit is different from that of the preceding request. (Current version ignores this field).

Code and Status

The Code and Status field has three parts:



Rightmost byte of word one in PP1 message buffer.

1. 4-bit level number

   a. Level number will be returned when an end-of-record occurred at a reading.
   b. Level number of the preceding logical record will be returned for a backspace logical record.**
   c. Level number must be specified for writing a logical record or a EOF**.

   ** Not available

2. 3-bit Error Code

   4 = order unacceptable.
   Three conditions will hang the system with relevant messages on the display:

   a. Uncorrectable parity errors.
   b. Uncorrectable lost data, and
   c. Failure to connect device.

3. 5-bit Order Code/Reply

Order	Code	Reply	Function/Status
Read	00	01 03 05 07	No EOR/EOF occurred. EOR occurred. End-of-File occurred. Tape Mark (End-of-File) read.
Write**	10 12 14	11 13 15	To write a full length record. To write a logical record. To write an EOF.

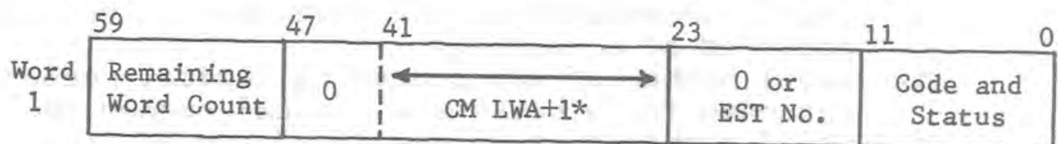
Backspace** PRU	20	21	To backspace one PRU
Backspace** a logical record	22	23	To backspace a logical record.
Rewind	30	31	To rewind to the load point.
Unload	34	35	To rewind and unload.

** Not available.

Execution of Orders

1. Read (00)

   Reading stops at a short record, tape mark, or when the remaining word count has been reduced to less than 1000B. When execution of the request is completed, information will be returned to PP1 message buffer word one as follows:



* CM LWA+1 indicates the CM location containing the level number if the record is a short record.

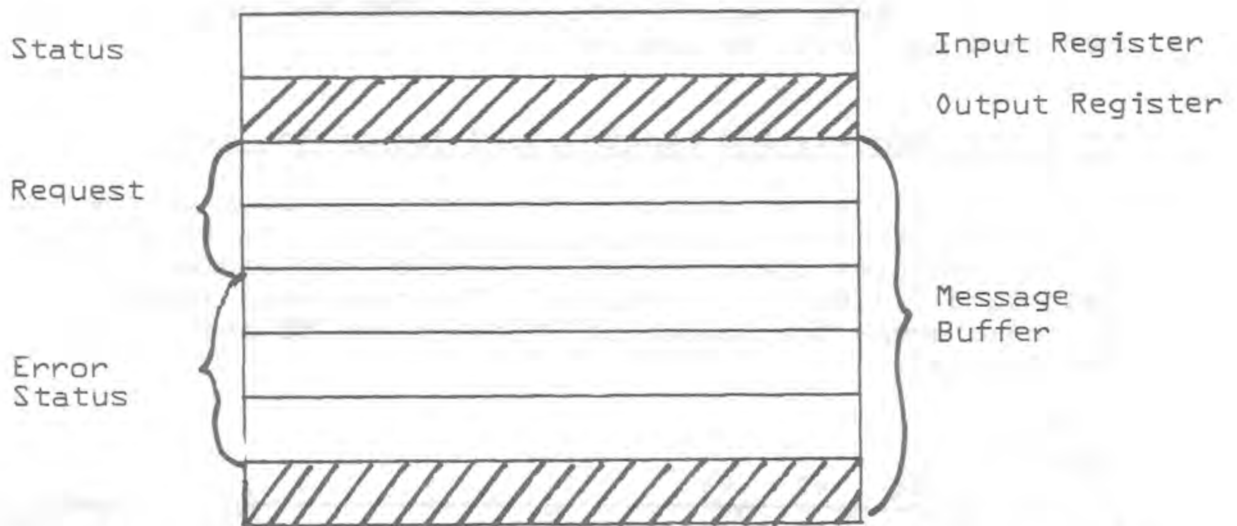
4.4.4.2 IRP - RMS Control

RMS Control resides in PP2 and processes all requests for mass storage I/O during dead start. RMS Control makes use of the drivers 5CP - 5CT as overlays to drive the appropriate device.

Status Communication

When RMS control is activated, it uses the PP2 communication area to exchange information with IRCP and other requesting routines. The last byte of PP2 input register contains status information, zero indicates a request is waiting.

PP2 Communication Area



Routines requesting RMS control activity place their request in words one and two of PP2 message buffer and zero the last byte of the input register. RMS control senses the "request waiting" status of the byte, executes the request and indicates completion by placing status information in the byte.

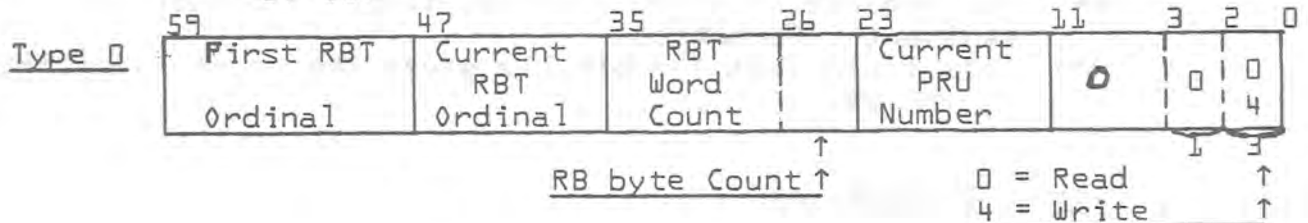
RMS control activities are terminated when the input register byte is set to 77B. At this time, PP2 enters into an idling loop in which it seeks a "two" at CM location zero.

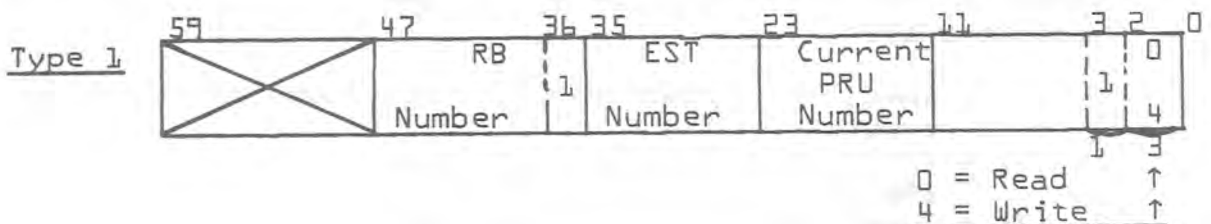
Types of Requests (Word One of Request)

There are four kinds of requests: Read and Write for each of type 0 and 1.

Type 0 - The request refers to an RBT chain.

Type 1 - There is no reference to an RBT chain and the request has RB number and EST number instead of it.



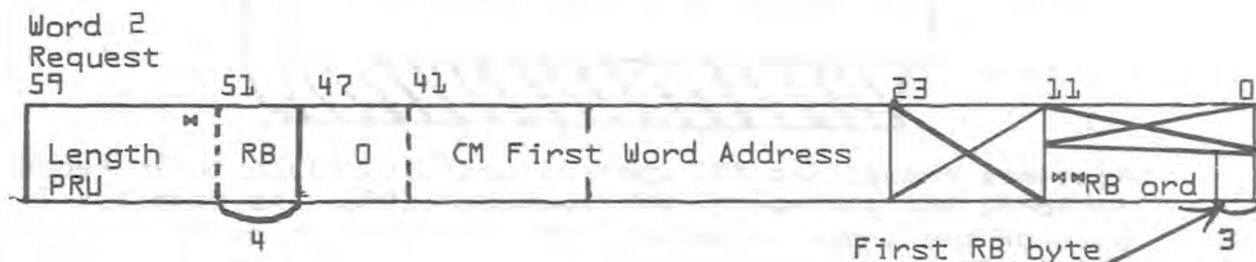


At the completion of a request, current PRU number will be updated to point to the last PRU+1. For type 0 request, current RBT/RB pointer may also be updated. For order 04 (write with RBT), end-of-information PRU No. held in first RBT word pair will be updated.

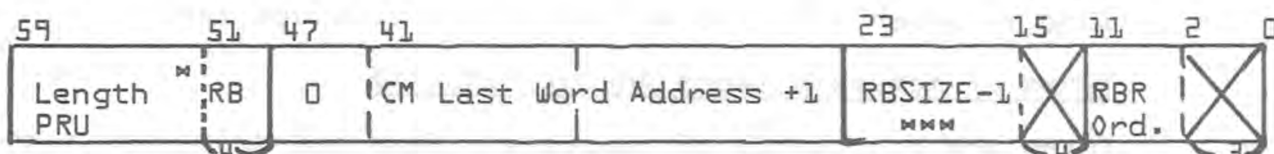
CM Buffer Address and Length (Word Two of Request)

Common to all kinds of requests, word two has a pointer to the CM buffer and its assigned length. At the termination of the request, these fields will hold; the address + 1 of last word filled by the request, the remaining length of assigned area, the number of PRU's in the RB and current RBR Ordinal.

Word 2
Request



Word 2
Return



* Length has two fields
RB (bit 0 - bit 3) gives number of full RB's to be processed, PRU (bit 4 - bit 11) gives additional length in number of PRU's.

** For Request with PRU = 7777B, this field must have relevant information.

*** This field (bit 4 - bit 11) gives the count -1 of PRU's in the RB.

Status Information

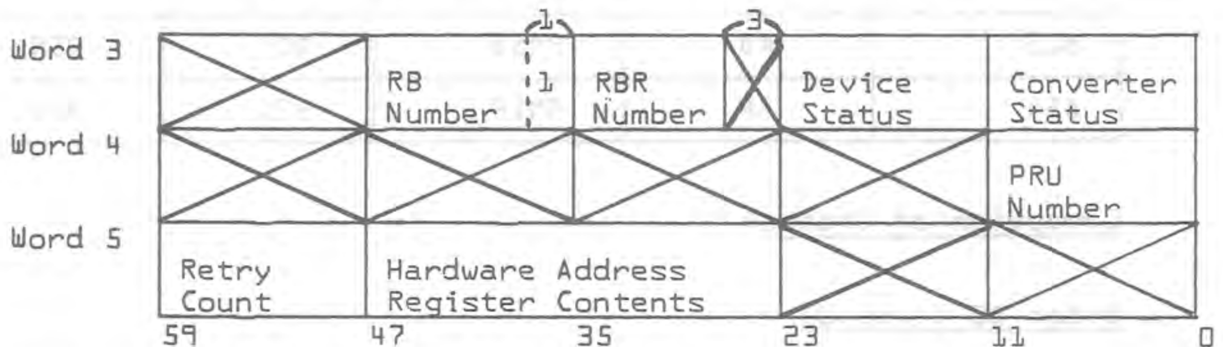
Kinds of status information put in the last byte of input registers are:

1. 1 = Specified number of PRU's have been processed without errors.
2. 41 = (RMS driver overlay not found
   (Request in format error.
3. 21 = Specified device not available.

4. 11 = Permanent parity error or lost data.
5. 05 = Recovered parity error or lost data. {This status bit {bit 2} can appear combined with other status information.}
6. 03 = {READ {00} ended with End-of-Information.
{WRITE {04} ended with current RBR becoming full
{READ {10} or WRITE {14} ended with the end-of-current RB.

For errors with code 21, 11 and 05, additional information will be available in word 3, 4, and 5 of message buffer as explained in the following.

Error Information {for status code 21, 11 and 05}



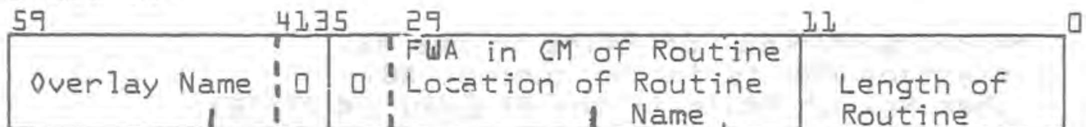
Directories for RMS Drivers

One-word directories for each of RMS to be used at dead-start time are assembled before RMS Control is to be used.

Their starting address is defined as DIRECT.

The end of directories is signaled by zero word.

The format is:



When a request is given to the RMS Control, RMS Control searches the necessary driver overlay from the directories if it is not currently loaded, and loads it using information on the directory.

RMS Driver Overlays

RMS driver overlays are assembled as segments of CONTROL.

They are assembled from common decks of UPDATE library {COMDECK} with a special assembly option for RMS Control.

A symbol RMSC is defined in the RMS Control for an assembly option to get correct driver overlays.

Device names and corresponding COMDECK names and overlay names are;

Device	Mnemonics	COMDECK	RMS Control Overlay	{ 1SP } {Overlay}
6603	AA	RMSA	5CP	3SP
6638	AB	RMSB	5CQ	3SQ
6603-II	AC	RMSC	5CT	3ST
865	AD	RMSD	5CR	3SR
854	AP	RMSP	5CS	3SS

Execution of Orders

Order 00 {Type 0}

READ WITH RBT

Reading starts at the current RBR/RB/PRU* specified.
Reading ends at;
a. a device error {other than recovered parity errors}
b. the end of RBT chain {PRU preceding EOI PRU}
c. the end of specified No. of PRU's

Switches from one device to another will be done within RMS Control.

*When 0 ≤ PRU < No. of PRU's in the RB.
starting PRU is in the current RB.
when No. of PRU's in the RB ≤ PRU {< 7777B}
current PRU is the first PPU {=0} of the next RB.

Order 04 {Type 0}

WRITE WITH RBT

1. First RBT Ordinal ≠ 0

a. PRU = 7777B
No RB is assigned for the new RBR specified in the request. RMSC will search the specified RBR for free RB** and if found toggle it and store the RBR No. and RB No. to the current byte of the RBT word pair. No actual writing will be done.

Then RMSC sets

PRU No. = 0
(Number of PRU's in RB) -1 in byte 9 of the request word pair.

b. PRU ≠ 7777B
Writing starts at the current RBR/RB/PRU* specified.
Writing ends at:
1. a device error (other than recovered parity errors)
2. the end of available RB's in the RBR
3. the end of specified No. of PRU's

** When the device is 6603/6603II, only outer zones will be used.

2. First RBT Ordinal = 0

a. PRU = 7777B
Lack of First RBT ordinal means no RBT word pair is assigned yet. RMS Control will get an RBT word pair from the empty chain and save it in the request format as the first RBT ordinal. Then last byte of word 2 of request will be put to the second byte of the word pair (as the RBR ordinal and First RB byte address). So, this information must be there when this type of request is to be issued. After that, the request will be executed as case (1a.) above.

RMS Control sets
1. Current RBT ordinal = first RBT ordinal
2. Current RB byte position = specified by the request.

Order 10 (Type 1)

READ DIRECT

Reading starts at the current RBR/RB/PRU specified.
Reading ends at:
1. a device error (other than recovered parity errors)
2. the end of current RB
3. the end of specified No. of PRU's

Order 14 {Type 1}

WRITE DIRECT

1. PRU = 7777B

   The specified RB bit of the RBR will be set and the PRU will be set to 0. The No. of PRU's in the RB-1 is returned.

   No actual writing will be done.

2. PRU ≠ 7777B

   Writing starts at the current RBR/RB/PRU specified.
   Writing ends at:
   a. a device error {other than recovered parity errors}
   b. the end of current RB
   c. the end of specified No. of PRU's

At the termination of a request, the request will be returned with updated information:

RBR/RB/PRU will point the next PRU to be processed.

No. of PRU's will show the remaining PRU's to be processed when the processing ended with an error condition.

CM FWA will point the address to be used next.

More specifically:
1. No change will be made when no processing has been done:
   PRU = 7777B and no available RB exists in the RBR.
   *Driver overlay not found.
   *Device not available.
   *Condition {b} or {c} {for any type} exists at the beginning.

*If PRU = 7777B Request will be modified and RB will be toggled before these errors are checked.

2. At a device error, the end status will show as if the current PRU {with error} has been processed, i.e., current PRU = PRU with error + 1, but CM address and the remaining PRU count will not be updated for the last PRU.

3. At condition {b} the current PRU will be the last PRU No. +1.
   for order 00 PRU = EOI in the RBT first word pair,
   for orders 04, 10, 14 PRU = No. of PRU's in the RB.

4.5.0 IRCP

This section describes in more detail the main functions of IRCP that were introduced in Section 4.2.8.2. The listing should be consulted for detailed descriptions of all entry and exit conditions.

4.5.1 Preliminary IRCP Tasks

IRCP first performs a few miscellaneous tasks that are easier to do here than in CED. For the most part, they have to be done before any of the major functions take place:

1. Central memory size (and ECS size in some cases) is saved.
2. For all RBR's of devices of fewer than 2048 record blocks, the unused bits are set.
3. The RBT area from the previous deadstart is saved for use by recovery.
4. The entire RBT area is initialized to a single empty chain.

The remainder of this section describes the major IRCP functions. See Section 4.2.8.2 for the order in which these functions are performed.

4.5.2 Preloading

The preloading and loading logic is much simpler than that of the previous version of deadstart. The reasons for this are:

1. The central processor has complete control, while the PPU's simply carry out requests. The RMS control routine in PP2 contains all logic required to drive all standard allocatable devices.
2. Because of the way central memory is handled, preloading and loading make direct use of the RBT and RBR areas. Previously it was necessary to place a special RBT chain on the first RB of the system device for use by loading. CMR was not loaded until preloading was complete. When loading was complete, another step had to be taken to build the RBT chain and set RBR bits for the system file.

During preloading, one PRU is read at a time from the system tape into one of the tape buffers, TBUF0 or TBUF1. The data is then formatted into allocatable device PRU's, stored into one of the RMS buffers, RMSBUF0 or RMSBUF1, and output to the device when the buffer contains one record block worth of data.

4.5.2.1 Tape Input

The system tape is rewound both at the start and at the end of preloading.

A tape PRU is read by issuing a request to the tape I/O routine in PP1. Only the first request issued (for each reel if a multi-reel system) requires preloader to wait for completion. For all subsequent requests, control passes directly to the processing of data in the other tape buffer. When it comes time to examine the results of the request, the following is done:

1. End-of-reel status is tested. If present, the tape is rewound, and a message is sent to the operator telling him to mount (or dial in) the next reel. After the types GO, the request is repeated.
2. End-of-file status is examined. If present, flags are set to cause preloader to terminate the next RMS buffer without waiting for a full record block.
3. Error conditions are not checked by preloader because they are checked by the tape routine in PP1.
4. If the tape PRU just read is the start of a logical record (if the previous PRU was short), the display is changed to show the name of the program currently being read from the tape.

4.5.2.2 Device Output

Once the input operation is completed for one of the tape buffers, the process of adding to one of the output buffers takes place. This is accomplished by the subroutine PLPRU, which is also responsible for outputting the buffer. Each time PLPRU is called, one PRU of data is placed in the output buffer. The size of a PRU, which is traditionally 64 CM words of data, is defined by the DSLCOM symbol (Section 6.0),RMSPRU. The following decisions affect the manner in which PLPRU is called and, hence, the logical record structure of the system file:

1. If there is more data in the tape buffer, and it is at least as much as a device PRU, PLPRU is called to place a full PRU in the buffer.
2. If there is more data in the tape buffer, and it is less than a device PRU, PLPRU is called to place the remainder of the data in the buffer. This produces a short PRU and, thus, the end of a logical record.
3. If the remaining word count of the tape buffer is zero, and if the tape record was not short, then this is not the end of a logical record. The processing of this tape buffer is complete.
4. If the remaining word count is zero, and if this was a short tape record, then this is the end of a logical record. However, the last PRU placed in the output buffer will have been a full one. Therefore, PLPRU is called to output a zero-length PRU. This is also done when the logical end-of-file is encountered on the tape.

Note that the above decisions make the assumption that the tape PRU size is an exact multiple of the device PRU size. Otherwise, in Case 2 above, a short tape record could not be assumed.

At the same time as the transfer across buffers is taking place, the other tape buffer is being read into by PP1, and the other RMS buffer is being output to the system device by PP2. With this double buffering process, the speed of preloading is limited only by the maximum rate of transfer to/from the slower of the two devices involved.

4.5.2.3 Completion of Preloading

There is very little cleaning up to be done at the end of the preloading process, since both the RBT chain and the RBR bits for the system file have been set up by the RMS routine. Only the following is done:

1. PLWAITR is called to wait for the last RMS request to complete. It is important that this be done before modifying the RBT's because there is no interlock on any of the tables during deadstart time.
2. The device type and the start address of the system file are stored in the FNT entry for SYSTEM.
3. The unfilled special fields in the system file RBT chain are completed. These consist of the random bit, the end-of-information PRU number, and the last RBT ordinal and byte number.
4. The tape is rewound.
5. The display is modified to indicate that preloading is complete.

4.5.2.4 Preloading Subroutines

4.5.2.4.1 PLDEV

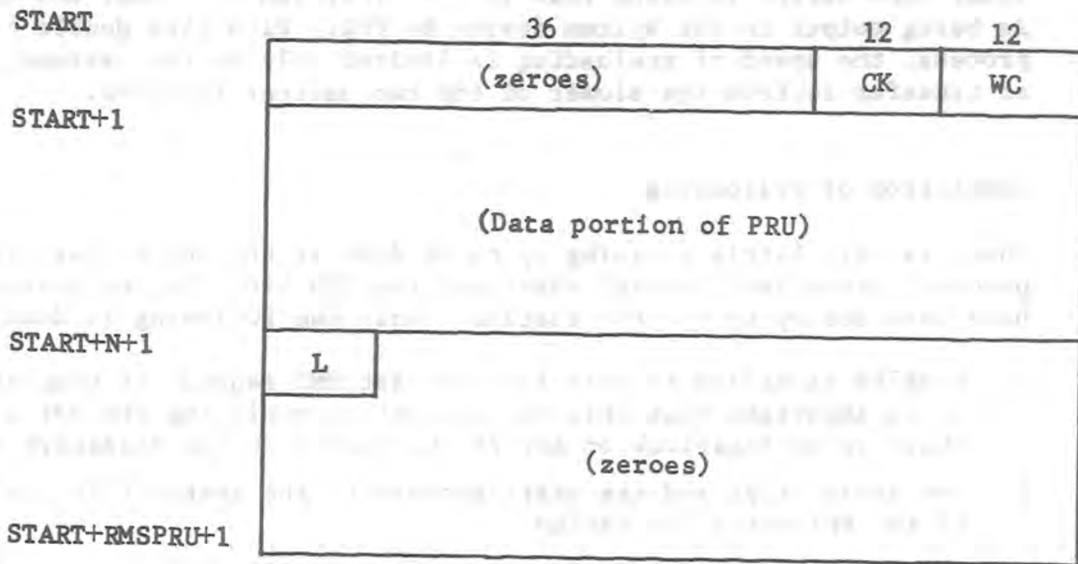
This routine is called to perform initialization whenever a new system device is needed for the first time. Normally, this is necessary only once at the start of preloading. Subsequent devices are required only if the previous device overflows. The following is done:

1. The EST ordinal for the next system device is picked up. If there are no more system devices specified or if five have already been picked up, deadstart terminates with an error message.
2. The display is modified to show the current system device being used for preloading.
3. The initial RMS request for the new system device is made. This request does not perform any I/O, but instead it allows the RMS routine to set fields in the RBT chain to indicate the switching devices.

4.5.2.4.2 PLPRU

This routine places data in the output buffers and issues RMS requests to write on the system device. Each time it is called, the equivalent of one PRU of data is placed in the buffer. The number of data words

to go in the PRU is passed on entry, and may be from zero to the size of a full PRU. The result appears as follows:



where N = Number of CM words of data in PRU.

$0 \leq N \leq RMSPRU$

CK = 12-bit checksum of all data words. This is the first byte actually written to the device by the driver.

WC = Number of data bytes in PRU (=5N)

L = Level number. This is present only in short PRU's ($N < RMSPRU$). It is equal to 17B for the EOF PRU and zero for all other short PRU's. (For the EOF PRU it will be in word START+1, since it is a zero-length PRU.)

After the PRU has been placed in the buffer, a check is made to see if the buffer now contains a full record block of information or if the PRU just stored was the EOF PRU (zero length and level 17). In either case, it is time to dump the buffer. First, PLWAITR is called to wait for completion of the last RMS request, then pointers are switched to indicate that the buffer just filled is now I/O active and the other buffer free for storing new data, and finally RMS control is called to begin another output request. PLPRU then exits while the request is still being processed.

4.5.2.4.3 PLMSG1

This routine simply makes a call to the display routine OPCOM to display the preloading message. It is called at the start of preloading and each time the program name or device information to be displayed is modified.

4.5.2.4.4 PLWAITR

This routine waits for the current RMS request to complete and then checks for error conditions. The conditions checked are:

1. RMS control found the request to be illegal, either because of a non-existant function code or the necessary driver was not found in the directory in CM. Neither of these conditions should occur.
2. Device reject.
3. Lost data (parity errors are not detected during writing).

4.5.2.4.5 PLTAPEF

This routine is called to issue either a rewind or unload request to the system tape. At the present time, the tape is never unloaded by deadstart.

4.5.3 System Loading

The system loading phase is the last main task of deadstart before the system routines take control of the computer. The necessary tasks are performed by the routine LOAD in IRCP and, finally, by STL. STL performs those tasks which can only be done in a peripheral processor and primarily involve starting up the other peripheral processors.

At the beginning of loading, the following is already done:

1. All information on mass-storage is already set up.
2. CMR is already set up except for a few values which must be stored by LOAD. If this was a recovery deadstart, the recovery process has already occurred. All RBT's are stored in their final location.
3. PP's 0, 1, and 2 are still a display driver, tape driver, and RMS driver, respectively. However, the use of PP1 as a tape driver ended when preloading completed.

From this, LOAD has to accomplish the following:

1. All remaining pertinent information has to be placed in central memory. This primarily involves setting up the library directory, which is not a part of CMR.
2. The appropriate system routine must be sent to each PP and given control.

4.5.3.1 Types of Loading

Loading is performed in one of three possible ways. The first is always used unless a recovery is taking place in which case, any one of the three methods might be used:

1. The entire system file is read from the device on which it resides. The resident library is built according to the Program Name Table on the system file. This restores the system to the state it was in before any EDITLIB's were run, in case this is a recovery.
2. The last copy of the library directory saved by EDITLIB (file SSSSSST) is read into the directory area. This has the effect of leaving the system as it was with all EDITLIB's except the most recent.
3. The Entry Point Table and Program Name Table are left as they existed prior to dead-start. Only those CM resident programs which have to be restored (because they are no longer intact in CM) are read from the appropriate file. This file is either SYSTEM or the EDITLIB-produced file, SSSSSSU. This type of load causes the system to be restored as it existed with all EDITLIB's.

4.5.3.2 LOAD I/O

The I/O performed by LOAD consists of input only, and is accomplished by issuing requests to RMS control in PP2. The exact method of reading the files SYSTEM, SSSSSST, or SSSSSSU varies with the type of loading being performed.

For the first type (Section 4.5.3.1), the file SYSTEM is read from beginning to end. As each PRU is encountered, it is examined to see if it is the start of a logical record, and, if so, the action performed for the remainder of that logical record depends on 1) the name and/or 2) the part of the system file currently being read. This procedure is also used for the second and third types on the system file through DSD.

The second and third types of loading do not involve reading the entire system file. Only certain records have to be read. This is done by setting up a request for RMS to read from a particular location in a file.

The following subroutines are used for all of the types of loading. Other, more specialized, routines are described later.

1. RDNXTRB
   This routine calls RMS to read a record block. The fields in the request which tell where on the file to read are not modified by RDNXTRB. They must be set up before RDNXTRB is called. For the first type of loading, this needs to be done only once, since SYSTEM is read sequentially, and RMS returns the request updated to the point where reading stopped.

   RDNXTRB waits until the request is completed. If the RBR ordinal in the completed request is different than the previous one, then the routine GETDEV is called to store values that change as a result of device switching. RMS errors are also detected by RDNXTRB.

2. CHNGREQ
   This is called by RDNXTRB to make the required settings in the RMS request prior to issuing the request. The FWA to read is set to CMBUFFER, and the desired read length is set to one record block.

3. GETDEV
   This is called by RDNXTRB whenever the input operation switches to a different device. The RBR ordinal, RB size, DST ordinal, EST ordinal, and EST address are saved by this routine. DEVINFO is called to set up the display line for the new device. RDNXTRB is initialized so that it will always call GETDEV the first time.

4. CHKEND
   This routine determines when the current RB is exhausted and time to read another RB. It does this by bumping the PRU number and comparing with the maximum.

5. GETPARAM
   This is called to fetch the header bytes from the start of the next PRU to be processed. These include the byte count and the PRU checksum. The fetch pointer for the RB is also advanced by one so as to skip over the CM word containing the header bytes. The format of each PRU after being read to CM appears in section 4.5.2.4.2.

4.5.3.3 Loading Procedure

1. The subroutine SETREQ is called to set up the initial RMS request to read starting at the beginning of SYSTEM. Then RDNXTRB is called to read the first record block.

2. The fetching of data from the input area is accomplished by first calling the routine CHKEND and then calling GETPARAM. By this process, the fetch pointer from the input area is updated to the next PRU, and, when necessary, the next RB is read. Except for the special processing of types 2 and 3 of loading, control returns to this point after each PRU has been processed.

3. If the current PRU is the end of the system file (zero-length, level 17), the routine ENDFILE (step 9) is entered. This only occurs here during type 1 of loading.

4. If the current PRU is the start of STL, the store pointer is set so the record will be stored at STLBUF. Then the routine PROCESSP is called. PROCESSP is used to store STL, MTR, DSD, the Entry Point Table, and the Program Name Table in their respective locations in CM. It sets a flag (MOVEFLAG) so that the data will be moved and then calls CHKSUM. Before exiting, PROCESSP sets a flag to indicate whether or not the PRU was short.

CHKSUM is the main data moving routine for the type 1 loading and also for types two and three through the processing of DSD. It does the following:

A. Picks up the PRU byte count and converts it to a CM word count.

B. If the data is to be stored in CM (MOVEFLAG = 0), the PRU is checksummed and, at the same time, moved to the area designated by the store pointer, STOREPT. The computed checksum is compared with the checksum in the PRU, and if they disagree, an error halt occurs.

C. If the data is not to be stored in CM, (MOVEFLAG = 1), only the checksumming is performed. This occurs for disk-resident programs.

The same process as described above is performed for MTR and DSD. MTR is stored at MTRBUF, and DSD is stored at DSDBUF.

5. If the current PRU is part of one of the records not used at all during loading, the routine IGNORE is entered. IGNORE acts much like PROCESSP, except it sets MOVEFLAG so that no data will be stored in CM. The records of this type include all dead-start records except those processed by PROCESSP.

6. After DSD has been processed, different branches are taken for each type of loading. If a recovery to "back up one EDITLIB" has been requested, processing continues at step 10. If the running system is to be recovered, processing continues at step 17. For the other types of recovery and for any intial or normal dead-start,processing continues at step 7 to perform a type 1 load.

7. The Entry Point Table (EPT) and Program Name Table (PNT) are processed in the same way as STL, MTR, and DSD. However, it should be noted that certain assumptions are made about the order of the remaining records on the system file:

   A. The Entry Point Table immediately follows DSD.

   B. The Program Name Table immediately follows the Entry Point Table.

   C. The library programs begin immediately after the Program Name Table, and they are in the exact order as designated in the Program Name Table.

8. The remainder of the system file consists of the library programs. Each program is processed as follows:

   A. For all PRU's except the first, control goes to step F.

   B. The next PNT entry is picked up from CM, the name is placed in the loading message, and OPCOM is called so as to display the name of the current program.

C. If the current program is the first non-PP program, a pointer to this PNT entry is stored in the word immediately before the start of the PNT.

D. The mass-storage address of the program is inserted into word 2 of the PNT entry as follows:

```
Bits 0-11    =   PRU number
Bits 12-14   =   RBT byte
Bits 15-23   =   RBT ordinal
Bits 24-35   =   RBT address
Bits 36-41   =   DST ordinal
```

E. If the program is to be disk resident (as indicated in the PNT entry), MOVEFLAG is set equal to one so that CHKSUM will not move the program to CM. If the program is CM resident, MOVEFLAG is set equal to zero so that the moving will take place. Also, if CM resident, the current value of the store pointer is placed in bits 24-41 of word 2 of the PNT entry. Word 2 thus contains the following for CM resident programs:

```
Bits 0-11    =   PRU number
Bits 12-14   =   RBT byte
Bits 15-23   =   RBT ordinal
Bits 24-41   =   Address in CM
```

F. The routine CHKSUM is called to dispose of the data in the PRU as described above. Finally, a flag is set indicating whether or not this was a short PRU.

9. The terminating procedure of LOAD is at ENDFILE and is entered for all types of loading. It does the following:

A. The LWA+1 of the library is stored in P.LIB.

B. The LWA+1 of the library is rounded up to the nearest 100B and is stored in each control point area in two places: 1) RA of the exchange package; and 2) The RA field in W.CPSTAT.

C. PP0 is signalled to stop driving the display. This is done by storing a 12B in CM location zero. It is necessary to wait for PP0 to pick up this signal because timing problems occur when STL gets too far along before PP0 gets out of its main display loop.

D. PP1 is signalled to load and begin execution of STL. PP2 is signalled to stop driving RMS control and to wait for its PP number to appear in CM location zero. PP's 3 through 9 have been doing this since early in the dead-start process.

E. If the BNL ECS code is assembled, it is entered at this point. Its functions are fully described in the BNL ECS section of the IMS.

F. The remainder of the loading process is the responsibility of STL, so IRCP stops. The procedure followed by STL is described in detail in section 4.2.9.

10. If a recovery was selected so as to restore the system as it existed prior to the last completed EDITLIB, the type 2 loading procedure is used. To accomplish this, the only loading performed is that of the file SSSSSST. This is the copy of the library directory as it was before the last EDITLIB. To find the starting address of SSSSSST, the routine SSSFNT is called. This routine is used for both type 2 and type 3 loads to search the FNT for an entry with a given name and of type COMMON.

11. If there is found to be no file SSSSSST, then no EDITLIB's have taken place. Control returns to step 7 to perform a type 1 load.

12. Using the values in the first word of the FST for SSSSSST, the RMS request is set up so as to read from the beginning of SSSSSST.

13. The store address is set equal to the FWA of the library directory. This value is obtained from P.LIB.

14. The routine LOADREC is called to perform the loading of SSSSSST. This routine is used for both types 2 and 3 loads. It does the following:

    A. Routine CHKEND and GETPARAM are called to handle the input of data. This is the same input process as performed at step 2. Note that the RMS request is already set up so as to indicate from where to read.

    B. If this is the first PRU of the record, the display is updated to show the name of this record. Also, the fetch pointer is advanced so as to skip the header words which are not to be stored in CM.

    C. The data in the PRU is moved to CM, and both the fetch and store addresses are updated.

    D. Steps A through C are repeated until a short PRU has been processed.

15. At this point, everything in the library directory is set up as desired. However, there is one problem area which has to be considered. The PNT contains RBT addresses for all disk-resident programs. These assume that the RBT chains for SYSTEM and SSSSSSU are in a certain location. However, since this was a recovery, it is quite possible that they are now in a different location than before, since Recovery moves all RBT chains to be saved up from the save area. Any given RBT chain may end up in a different location because of other chains for scratch files which were located higher in CM, and which are discarded. It is quite possible for this to happen to SSSSSSU.

    To take care of this, it is now necessary to determine the RBT address for each program by using 1) the RBT ordinal for the program, and 2) the first RBT address as given in the FST entry for the respective file.

16. To terminate the loading process, control now transfers to the routine ENDFILE at step 9.

17. If a recovery was selected so as to leave the system as it was after all EDITLIB's, it is almost possible to not have to do anything about setting up the library directory area. This is because the library directory as is now desired was in CM at the time of dead-start. However, it is likely that the CM resident programs extend past the point where dead-start uses for buffer areas during label processing and preloading. Therefore, it is necessary to read in all programs which are not still intact in CM. This is method 3 of loading.

    If the PNT is so large that it extends past the point described above, then this type of loading cannot be performed, and an error halt occurs. However, for 65K and 131K users, the danger of this ever occuring is forever overcome by altering certain symbols in DSLCOM. (See section 4.3)

18. Each PNT entry is examined. Step 19 is performed only for those programs which are CM resident and extend beyond the area which dead-start has used for a buffer (TBUFO, location 15700B). Actually, step 19 could be performed for CM resident programs, but since reading of the system file in this manner is much slower than with type 1 of loading, it is done only when necessary.

19. The routine SSSFNT is called to find the FNT/FST entry for either SYSTEM or SSSSSSU depending on which file the program resides. Using the first RBT address in the FST and the RBT ordinal, RBT byte, and PRU number in the PNT entry, the RMS request is set up so as to point to the start of the program. The routine LOADREC is now called to read the program and move it to CM.

20. Steps 18 and 19 are repeated for each PNT entry. When this is completed, control goes to step 5, and the remainder of the loading process is identical to type 2.

4.5.4 DEADSTART RECOVERY

Recovery is called to recover files following a system failure. It searches the FNT and returns the system to an "initial state", i.e., all the jobs in the input queue are left there (except when the no rerun bit is set) and all jobs running at control points are rewound and placed back in the input queue. All files in the output queue are left there and all files in the process of printing and punching are rewound and put back in the output queue. All common files are disassociated from control point and put back in the common pool and all local files are released. Local dayfile FNT entries are initialized. The system dayfile and error file buffers are restored.

Initially, RECOVERY ascertains whether to restore only the system file, only other files (not the system file), or all the files. Then, Re-

covery checks the RBT channel to see if it was reserved at the time of the hangup. If it was, then the message RECOVERY RISKY is displayed on the left screen. The operator can type a GO if he wishes to continue. Recovery then puts a + before the name of the systemso that all successive dayfiles will reflect the fact that at least one recovery has taken place. The, the entries in the FNT are restored (excluding the entries for the subdirectories and RBTC which have already been restored by the permanent file portion of IRCP. The ON/OFF bit is restored for the entries in the Equipment Status Table and private packs are set to an unloaded status. The dayfile and error file buffers are restored. RESPOND files are recovered in the following manner: If the file has a disposition code of 2000_8, then it should have its entry in the FNT zeroed out but should retain its RBT chain. If the file has the name RSP,SYS then the active bit in the second word of the FNT entry should be checked to see whether RESPOND was active at the time recovery was initiated. If it was active, then the recover bit in the FNT entry will be set and the RBT chain will be restored. Checkpoint and locked files are then recovered. Recovery proceeds by searching through the FNT table and by performing the necessary action on each file. When a file should be saved, its RBT chain will be restored and the appropriate bits set in the RBR's.

At the conclusion, Recovery will go to PRELOAD if the system file has not been restored. Then it will exit normally. Since RECOVERY is a subroutine, the exit is to its return jump call.

4.5.5 Label and Permanent File Processing

4.5.5.1 General

The label and permanent file portion ofdead-start resides in IRCP and always performs its functions before preloading, loading, and recovery. It is responsible for determining which record blocks on each device are already allocated, either as part of permanent information (that which survives across dead-starts) or as flaws. To so indicate to the remainder of the dead-start process and to the running system, the RBR bits for all such record blocks are set at this time. It should be noted again the the fresh copy of CMR has already been loaded at this time.

Labels are put on every allocatable device in order to be able to determine which device contains the Permanent File Directory (PFD) and Record Block Catalog (RBTC). The device containg this information is only commonly known as the primary permanent file device. Note that the actual permanent files themselves may reside on any device in the configuration. Labels also carry information telling which record blocks are flawed.

The PFD and RBTC contain everything necessary to recover all permanent files at dead-start time. However, much of the information they contain is not used by dead-start. The specific formats of these tables may be obtained from the permanent file documentation.

During an initial-type dead-start, labels are written on all allocatable devices except those which are turned off, and an empty PFD and RBTC are written on the device designated as the primary permanent file device. All of this is done under close operator control. The various type-ins are described in the operator's guide.

During every type of dead-start in which IRCP is used (initial, normal or recovery), every device label is read, flaw information is moved to CMR, and the PFD and RBTC are read and all central memory tables required for permanent files are built.

A general description of all the major routines used for label and permanent file processing follows. Note that all functions related to permanent files need be performed only if it is intended to use permanent files. The assembly of such code is dependent on the value of the symbol IP.PFM.

4.5.5.2 LF - Main Routine

This is the routine called at the end of the preliminary IRCP processing to perform all of the label and permanent file processing. The following procedure takes place for all standard allocatable devices provided they do not have the ON/OFF bit set. Devices which are off will be processed, however, if the LBL bit is set in the EST (bit 57). The LBL bit will be set only as a result of using the LBL type-in while making equipment changes. The LBL bit is always cleared at this point. Label processing is never performed on ECS. The devices are processed in the order they appear in the EST.

1. If this is an initialization-type dead-start, a label is written on the device. If necessary, the permanent file tables are also written. All of these functions are accomplished by calling the routine IDEV.
2. RDLBL is called to read the device label. The label has to be read successfully or else the operator will be asked what to do. The specific operator options for each error condition may be found by consulting the listing of IRCP. In general, the options include trying again, writing a new label (step 1), or deleting the device from the system.
3. A check is made to see if the device contains a private pack. This is ascertained by a field in the label. If it is a private pack, the operator is informed. It has to be assumed that it was left mounted on the device by mistake, and that it is really not desired to have its contents destroyed.
4. The flaw bits in the label flaw table are placed in the appropriate RBR in central memory.
5. If the label indicates that this is the primary permanent file device, the routine PFB is called to read and process the PFD and RBTC.

After the above steps have been performed on all devices, a check is made to see if the primary permanent file device was encountered. If not, the operator is informed. If he types GO, processing continues,

and the routine LF exits. This is entirely proper if the configuration had been initialized without a primary permanent file device (and hence, no permanent files). However, this check is made to safeguard against the case where, for some reason, the primary permanent file device was not processed, but the other devices, which can certainly contain part of the permanent files, had been processed. If this were to happen, there is no way to tell which record blocks on these other devices contain permanent files, since this information is obtained from the RBTC.

4.5.5.3 IDEV - Initialize a Device

This routine handles label writing for one device and then, if necessary, gets an inital empty PFD and RBTC written. During an initialization-type dead-start, it is called by LF for every device and is also called by LF if certain error conditions occur where the operator has specified that a new label be written on a particular device. The following takes place:

1. If the operator has responded to the write option (step 8 in this sequence) for a previous device by typing NONE, then none of the IDEV functions need to be performed. The exit is taken immediately.

2. It is desired to know whether or not there is already a label on the device. The routine RDLBL is called to try to find a label.

3. If a label was found, and the label is that of a private pack, the operator is informed. He may either bypass the initialization procedure for this device, or have a non-private label written.

4. If the operator has responded to the write option (step 8) for a previous device ty typing ALL, then the label options display is bypassed in its entirety. Processing advances directly to step 8c.

5. The label options display is issued. It contains the following:

   a. Information to identify the device. (See description of DEVINFO).

   b. A message that tells whether or not the device already has a label. If it does have a label, the RB and PRU number of the label are given.

   c. Instructions for changing the variable fields in the label. This includes the default information which will be put in the label if no changes are specified. If a label was found on the device, the default information is obtained from the corresponding fields in that label.

   d. Instructions for use of the write options. When the operator specifies one of these, IDEV completes the processing for this device.

   e. A display of the contents of the entire flaw table. Again, this is obtained from the existing label, if present.

6. When the operator has made an entry, the entry is identified and processed as described in the following steps. If any kind of error is detected, the word "ignored" will appear at the bottom of the left screen. This will disappear as soon as a good entry has been made.

7. Label field entries cause the appropriate information to be changed both in the label buffer and in the display. Any number of entries of this type may be made. They allow the following:

   a. Changing the date carried on the label.

   b. Changing the visual identification carried in the label.

   c. Indicating whether or not this is the primary permanent file file device.

   d. Setting or clearing flaw bits.

8. There are four possible write control options in which any one of them signals the end of input for this label. They are as follows:

   a. SK - The operator does not want to write a new label on this device. IDEV simply exits.

   b. NONE - The operator wants label writing to be bypassed on this device, and on all devices yet to be processed. A flag is set and IDEV exits. This is the flag which is checked in step 1 above. This has the same effect as typing SK here and for all remaining devices.

   c. WR - The operator wants the label, as the display now shows it, to be written. This involves the following steps:

      i) If this device has been designated as a primary permanent file device, but there has already been such a device specified, the operator is told that this device must not be the primary permanent file device. There simply cannot be more than one. If the operator indicates for processing to continue, then this device will not be a primary permanent file device.

      ii) Flaw bits as indicated in the label area are transferred to the RBR in central memory.

      iii) The routine WRLBL is called to write the label on the device. If an error occurs, the operator is asked to respond in the same manner as with label reading in LF. However, there is one option here that is different than with label reading: designating that the label be written on the next record block. If this option is selected, a flaw bit is automatically set in the flaw table to indicate as unusable the record block on which the label write was unsuccessful.

      iv) If this is indicated as being the primary permanent file device, the routine PFA is called to write the empty PFD and RBTC.

d. ALL - The operator wants the label written on this device, according to the options he has set, if any. In addition, he wants a label to be written on all remaining devices without any more typeins required on his part. This yields exactly the same results as if he had responded to the display for each remaining device by only entering WR. To process the entry of ALL, the flag which is examined in step 4 is set. Then the processing is identical to that for the WR option.

4.5.5.4 RDLBL - Read Label

This routine is called to find and read device labels. The subroutine RWPRU is called to perform the actual setting up and issuing of requests to RMS control. RDLBL handles the logic for searching the device for a label.

The following information is supplied to RDLBL when it is entered:

1. EST ordinal of the device
2. The number of record blocks to search before giving up.
3. The record block to search first. Succeeding record blocks, if necessary, are picked from the next available in the RBR.

Two symbols which are defined in DSLCOM (Section 6.0) are of special interest here. RBLIM is defined as the maximum number of record blocks to be searched on any one call to RDLBL. PRULIM is the maximum number of PRU's which will be searched in each record block before giving up on that record block. The values of these may be modified to suit an installation's desires, but for any reliable device, both should be defined equal to one.

The following conditions are required for a successful read:

1. The RMS read request must be completed without errors.
2. The expected label ID must be found at the start of the label.
3. The date field must be present, and must be at least as recent as the minimum allowable date. This is to avoid possible conflict with some type of label written by somebody other than dead-start. All labels written by dead-start (IDEV) meet this condition.
4. The checksum must be correct.

4.5.5.5 WRLBL - Write Label

This routine handles the writing of labels on devices. When called, this routine is told the device and the record block number on wich to write the label. Then entire label contents except for the checksum are expected to be set up in the label buffer (LBLADR). The following procedure is used to make sure a good label is written:

1. The checksum of all the label information is computed and stored in the label.

2. RWPRU is called to issue an RMS request to write the label on the device at a certain PRU.

3. If RMS control returns an error status, then it is concluded that a label cannot be successfully written on this PRU. A call is made to RWPRU to write the same PRU, but this time a buffer of all zeroes is written. This is to prevent RDLBL from possibly mistaking this PRU for a good label.

4. If the designated limit of PRU's to try (PRULIM) has not been reached, step 2 is repeated. Otherwise, WRLBL exits with an error indicator. WRLBL must be called again in order for the label write to be attempted on another record block.

5. If there was no error status in step 3, then RWPRU is called to read the label. If any error status is returned, or if the data read does not compare with the data written, then the above error procedure is taken; otherwise, the normal exit is taken.

4.5.5.6 PFA - PFD and RBTC Initialization

The routine PFA writes an empty Permanent File Directory (PFD) and Record Block Catalog (RBTC) on a device. It is called by IDEV during processing of the device designated by the operator as the primary permanent file device. This requires the following steps for both the PFD and the RBTC:

1. The number of record blocks needed to contain a PFD (RBTC) of the required length is computed. The following values from CMR are used in this calculation:

   a. Byte C.SDTL of P.PFM1 - the number of subdirectories in the PFD.

   b. Byte C.SDL of P.PFM2 - the number of entries in each subdirectory.

   c. Byte C.RBTCL of P.PFM2 - the number of PRUs/16 in the RBTC.

2. Thr routine GPFRB is called as many times as the number of RB's computed in step 1. This routine finds the next free RB and writes out that RB completely zeroed out. It repeats this process if necessary due to a write error. It also adds the RB number to a chain which will be written at the start of the first RB.

3. The first RB of the PFD(RBTC) is written again, but now containing the appropriate header and the list of RB numbers formed in step 2.

4. If a write error occurred in step 3, it is necessary to set up one more zero-filled RB and then write the header information on what was originally the second RB of the PFD (RBTC). The first (bad) RB is recorded as a flaw.

5. The pointer to the start of the PFD (RBTC) is inserted in the label information. This consists of the first RB and PRU numbers. The PRU number is always zero, since both the PFD and RBTC always start at the beginning of an RB. Note that the contents of the label output buffer has not been disturbed since the label of this device was written by IDEV.

6. After both the PFD and RBTC have been set up by the above steps, the device label is rewritten so that it will contain the PFD and RBTC pointers which were not known when the label was first written.

7. PFA exits to IDEV. No additional information has to be placed in the PFD or RBTC at dead-start time. This is done at the time permanent files are cataloged.

4.5.5.7 PFB - PFD and RBTC Checking

PFB is called by LF after the label has been read on the primary permanent file device. It is called on every dead-start of type normal, initialization, or recovery because its tasks must be performed if permanent files are to be preserved. The following information in CMR is set up:

1. FNT entries for each subdirectory of the PFD and one FNT entry for the RBTC.

2. One RBT chain each for the PFD and the RBTC.

3. RBR bits for all record blocks containing permanent information.

4. Subdirectory Table (SDT) entries giving the number of entries in each subdirectory.

This information is set up by doing the following:

1. The first RB of the PFD is read and the RB chain for the entire PFD is obtained. From this, an RBT chain is built in CM for the PFD. All corresponding RBR bits are set at this time.

2. An FNT/FST entry is stored for subdirectory zero. Subdirectory zero contains only the RB chain and is used only by dead-start. Therefore, the next step is not performed for it.

3. The remainder of the PFD is read and the number of active permanent file entries in each subdirectory is stored in the appropriate slot in the SDT. Also an FNT/FST entry is built for each subdirectory.

4. Step 1 is repeated for the RBTC.

5. The FNT/FST entry for the RBTC is stored in CM.

6. The entire RBTC is read. The RBT chains for all active permanent files (part of the RBTC) are examined, and all RB's allocated to permanent files are so indicated by having the appropriate bit set in the RBR tables. During the process of setting RBR bits for permanent information, both here and in step 1, a check is made whenever a bit is set to see if it was already set. If so, there is a problem somewhere. A message is displayed which shows the RBR and RB number with which this occurred. If the operator wishes to continue, he may do so by typing GO.

The following subroutines are used by PFB to perform the reading of the PFD and RBTC.

1. PFBREAD reads one record block by calling READRB. It is told which RB to read. See section 5.6 for the description of READRB.

2. PFBRRR is called to read one RB of either the PFD or RBTC. It determines which RB to read by using the appropriate RBT chain which by now, is already placed in CM. Once the RB number is obtained, the routine PFBREAD is called.

3. NXTWORD is used while scanning through the RBTC. It fetches the next word from the buffer into which one RB of the RBTC has been read. It returns the word and a status to indicate if the word is either the start of an RBTC entry, the end of the filled portion of the RBTC, or neither. Whenever the buffer is found to be exhausted, the routine PFBRRR is called to read the next RB.

4.5.6 IRCP General Subroutines

4.5.6.1 GETRBR

Searches the RBR area to find the RBR corresponding to a particular EST entry. Both the RBR ordinal and address are returned.

4.5.6.2 GETWDPR

Allocates space for and sets up an RBT word pair for an RBT chain. The empty chain is adjusted accordingly, and, if this is not the first word pair for the chain, the chain link field in the previous word pair is set to point to the new word pair.

The RBR ordinal, byte, and primary allocation type fields are set to the appropriate values. All other fields are set to zero.

Return is made with the RBT address of the new word pair.

4.5.6.3 CKCHAIN

Checks the length of the empty chain. If it is less than 100B words, it is increased by 100B words.

4.5.6.4 WAITGO

Checks to see if the operator response was GO. This routine is intended to be called immediately after a call was made to OPCOM to return alphanumeric data.

4.5.6.5 SQUANK

Sets zero characters to blanks for insertion into displays.

4.5.6.6 DEVINFO

Sets up pertinent information about a particular device for display purposes. The information consists of the EST ordinal, device name, channel number, equipment number, and unit number. Nearly every message directly related to a device contains the information produced by this routine.

4.5.6.7 DEVERR

Sets up the error information returned by RMS control about a particular device. A display is formatted from the information and consists of the RBR number, RB number, PRU number, device status, converter status, and hardware address register.

4.5.6.8 FINDBIT

Is given a record block number and returns the corresponding bit position and address in the RBR of that record block.

4.5.6.9 GETRB

Finds the next available record block in an RBR.

4.5.6.10 SETRB

Sets the bit in an RBR for a given record block number.

4.5.6.11 WRITERB

Writes one record block of information to a given device on a given record block. Since this routine is primarily intended for writing of permanent file information, the record block is checked by having the routine READRB read it back. WRITERB is not used by the preloader.

4.5.6.12 READRB

Reads one record block. On entry, the device, record block, and CM address are specified.

4.5.6.13 RWPRU and RWRB

Sets up and issues RMS requests to read or write full record blocks or single PRU's. This is a single routine with two entry points.

4.5.6.14 REJ

Displays a message to indicate the device last attempted to access has rejected a function. This is more likely to occur as a result of there being no such devices configured as such. Unfortunately, in many cases where a non-existent device is referenced, this routine will never get called because the I/O driver will hang on the channel and will never be able to return the error indicator.

4.5.6.15 DATEFJ

Converts a YYDDD Julian date to an MMDDYY date for use by the device label routines.

4.5.6.16 DATETJ

Converts an MMDDYY date to a YYDDD Julian date for use by the device label routines.

4.5.6.17 CNVTDTB

Converts a display coded decimal number to binary.

4.5.6.18 CNVTBTD

Converts a binary number to display coded decimal. Note that similar routines exist in the OPCOM portion of IRCP for converting from and to octal display. These are CNVTBTO and CNVTOTB.

4.5.7 OPCOM - Operator Display Package

The IRCP portion of OPCOM is called in order to process OPCOM macros. The OPCOM description in section 4.8 describes it in more detail.

4.5.8 BNL ECS Code

After dead-start loading (section 4.5.3) is completed, the BNL ECS code is executed. There is also BNL ECS code in STL which is used to drive the display. The assembly of this code is conditional upon the symbol IP.ECNOM being non-zero.

The BNL ECS code accepts information from the operator as to whether ECS is to be used or not, and if so, how it is to be allocated. For a detailed description, refer to the BNL ECS portion of the IMS.

4.6.0 Dead-Start Common Decks

The following common decks are used by one or more of the dead-start routines:

1. IPARAMS - Installation Parameter Definitions

   This is called by all of the dead-start routines: CEA, CONTROL, IRCP, and STL.

2. DSLCOM - Dead-start Definitions

   This common deck contains definitions of symbols used exclusively by the dead-start routines. Included are central memory addresses of buffer areas, device related information such as PRU sizes, symbols controlling the assembly of debugging code, and other similar information. Several macros also are in DSLCOM which have to do with information about the different device types. They are used to generate the various tables the routines used, and, thus, keep the routines independant of device changes. DSCLOM is called by CONTROL, IRCP, and STL.

3. DSMAC - Operator Communication Macros

   This common deck contains the definitions of all the macros used by OPCOM. More details are given in the OPCOM section. It is called by CONTROL and IRCP.

4. ECSCOM - BNL ECS Symbol Definitions

   This deck is called by IRCP and STL for use by the BNL ECS code. It is described in detail in the IMS section for BNL ECS.

5. RMSA, RMSB, RMSC, RMSD, RMSP.

   These are called by the dead start drivers (5CP, etc.) and the stack processor overlays (3SP, etc.) and contain all of the code used in common by the two types of driver overlays.

4.7.0 Hardware Aspects of Dead Start

4.7.1 Introduction

The dead start process requires that a short program (up to 12 PPU instructions) be set up on the matrix of toggle switches on the dead start panel. When the dead start switch is toggled, this dead start program is transmitted to PP0's memory and executed. This program reads the first record (CEA) from the system tape and transfers control to it. When CEA has finished execution, control is transferred back to the dead start (panel) program to read the next record (CED) from the system tape. From this point CED and other system dead start programs read the remaining records on the system tape and initiate system execution as described elsewhere.

4.7.2 The IAM Instruction

A detailed understanding of the dead start loading process requires some familiarity with the functioning of the IAM instruction. The IAM instruction is a 24-bit instruction: the d portion of the instruction holds the channel number and the m portion of the instruction contains the address in peripheral processor memory where the first data word is to be stored. The A register is assumed to contain the number of words to be read. The functioning of the IAM instruction is show in figure 4.7.1. Note the following points:

- During execution of the IAM instruction, the contents of the P register are stored in location 0, and the P register used to hold the memory address for the next word to be stored. At the time the contents of the P register are stored, P holds the address of the second word (m portion) of the IAM instruction. Before exiting the instruction, the contents of location 0 are read, incremented by one, and placed in the P register to provide the address of the next instruction.

- The IAM instruction tests the word count in the A register to see if it has been reduced to one: if so, (A) is reduced by one and the instruction exited. Therefore, if the IAM instruction is entered with the contents of the A register equal to zero, the word count is effectively 77777B.

- The IAM instruction may be exited in one of two ways: (1) because the word count has been reduced to zero or (2) because the channel has become inactive. If the word count has not been reduced to zero and the channel is active, exit will not take place even though no data is being read: the processor will idle in trip 4, waiting for the channel to become full.

4.7.3 The Dead Start Sequence

When the dead start switch is toggled, the following sequence is initiated:

- The Master Clear signal is generated.

- The A register of each peripheral processor is set to 10000B: the P register of each peripheral processor is set to zero.

- The K register of each peripheral processor is set to 712 (trip 4 of an IAM instruction).

- All channels are set to empty and active.

- All peripheral processors are connected to their respective channels (i.e., PP0 to channel 0, PP1 to channel 1, etc.) by setting the appropriate channel number in each processor's Q register.

The IAM Instruction

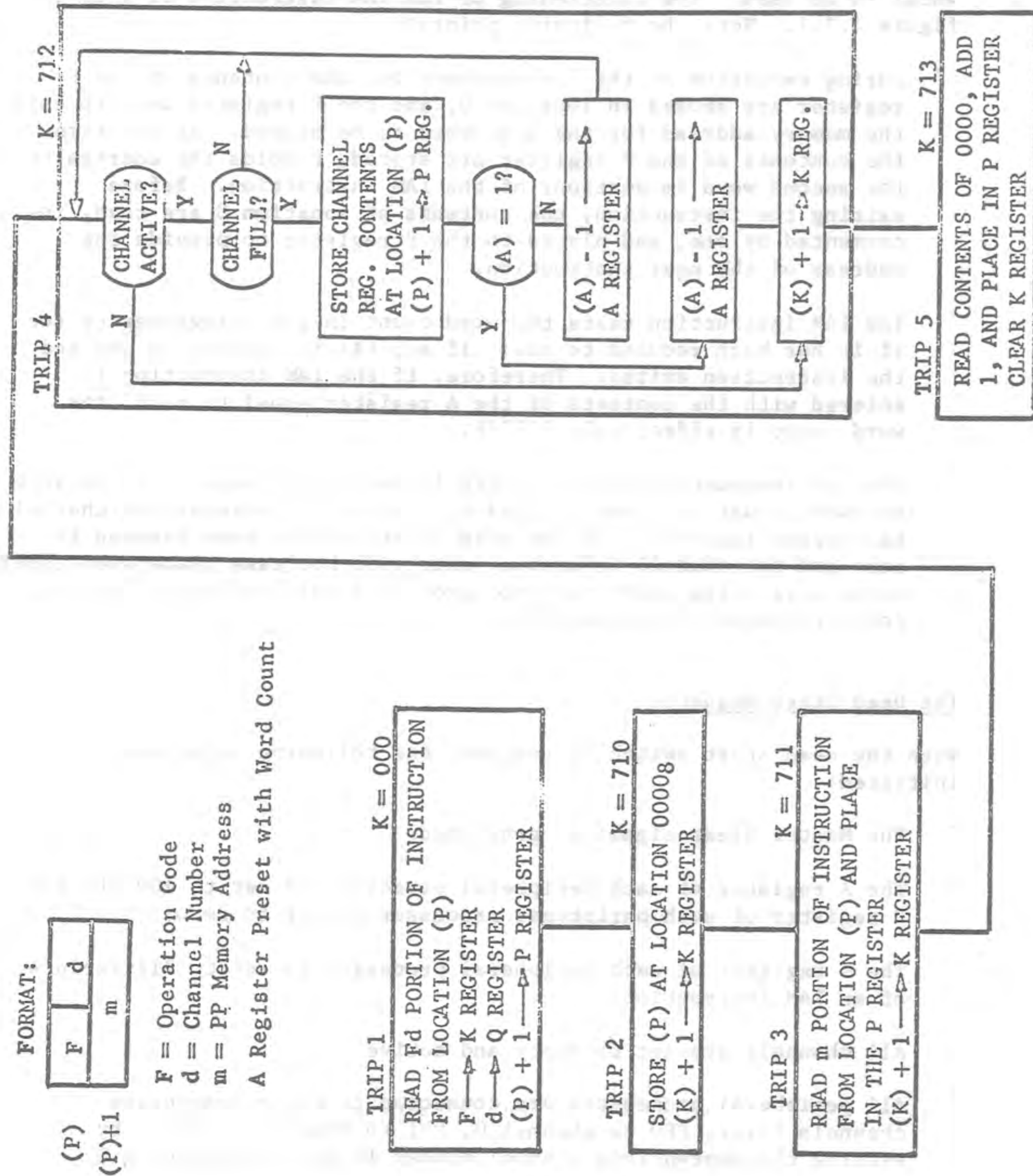


Figure 4.7.1

- The first synchronizer on each channel is selected: the first unit on that synchronizer is selected.

- The dead start synchronizer is selected on channel 0.

- The program on the dead start panel is transferred to PP0 memory: First, a zero byte is transmitted (stored in location 0); next, the 12 bytes from the panel switches are transmitted (stored in location 1-14); finally, another zero byte is transmitted (stored in location 15).

- The dead start synchronizer disconnects channel 0, initiating the execution of the dead start program.

Peripheral processor zero treats the data sent by the dead start synchronizer as it would data arriving from any other controller. When the dead start synchronizer disconnects from channel zero, peripheral processor zero exits from the IAM instruction. In exiting, the contents of location 0 are incremented by 1 and used as the address of the next instruction. Since this location was cleared to 0 by the dead start process, the address of the next instruction is 0001: this location holds the first instruction of the program sent by the dead start synchronizer from the dead start panel.

4.7.4 The Dead Start Program

A typical dead start panel program for a tape on channel 12 or 13 is shown in Figure 4.7.2. This program performs the necessary function issuing, activating, and input to read the first record of the system tape into PP0 starting at location 13B. (Notice that the settings of words 13 and 14 of the panel are immaterial.) When this record, which contains CEA, has been read the controller disconnects the channel upon detecting the end-of-record gap and PP0 exits from the IAM instruction. Following the IAM PP0 executes the contents of word 13 which is the first word of CEA.

A typical dead start panel program for a tape on channel 1-11 is shown in Figure 4.7.3. It does the same as above with one exception. To free the tape channel XX for use, words 13 and 14 are sent to words 0 and 1 of PPXX (actually the entire contents of PP0 is sent to PPXX since (A) = 0). Disconnecting the channel causes PPXX to input on channel 12 instead of channel XX. Channel XX is now free to use in reading the system tape.

4.7.5 System Dead Start Programs

The first dead start program read from the system tape is CEA. The first word of CEA, loaded at 13B, is a header containing the name of the program itself. This header word is actually executed following exit from the IAM as indicated above. But notice that word 13 will contain the first two characters (in display code) of the program name, or 0305. Interpreted as an instruction this is UJN 5 and so control transfers to word 20 where the body of CEA begins.

DEAD START PANEL PROGRAM

(Tape Channel = 12 or 13)

PPO Memory Location	Instruction	Mnemonic	Description
0000	0000	PSN	(Address-1 of First Instruction)
0001	75XX	DCN XX	Disconnect Tape Channel
0002	77XX	FNC EOUU,XX	Select Tape Unit
0003	ECUU		
0004	77XX	FNC 0010,XX	Rewind Tape
0005	0010		
0006	77XX	FNC 140S,XX	Select Input to End of Record
0007	140S		
0010	74XX	ACN XX	Activate Tape Channel
0011	71XX	IAM 0013,XX	Input Tape Record
0012	0013		to location 13B
0013	0000	PSN	Not executed
0014	0000	PSN	Not executed

XX = Tape Channel
E = Equipment Number of Tape Controller
C = CMR Skip Count

UU = Tape Unit
S = PPO Save Switch

Figure 4.7.2

DEAD START PANEL PROGRAM

(Tape Channel = 1-11)

PPO Memory Location	Instruction	Mnemonic	Description
0000	0000	PSN	(Address-1 of First Instruction)
0001	73XX	OAM 0013,XX	Free Tape Channel
0002	0013		
0003	75XX	DCN XX	Disconnect Tape Channel
0004	77XX	FNC EOUU,XX	Select Tape Unit
0005	ECUU		
0006	77XX	FNC 140S,XX	Select Input to End of Record
0007	140S		
0010	74XX	ACN XX	Activate Tape Channel
0011	71XX	IAM 0013,XX	Input Tape Record
0012	0013		to location 13B
0013	0000	PSN	Hang on input
0014	7112	IAM 0000,12	from channel 12

XX = Tape Channel
E = Equipment Number of Tape Controller
C = CMR Skip Count

UU = Tape Unit
S = PPO Save Switch

Figure 4.7.3

When CEA completes its function (explained elsewhere) control is transferred to word 6 (remember the dead start panel program is still intact) to read the next record from the system tape. This next record, which contains CED, is read starting at word 13B and control is transferred to CED in the same manner. From this point the dead start process is directed by CED and other system dead start programs. Control never returns to the dead start panel program.

4.7.6 Channel Usage During Dead Start

Remember that after the dead start switch is toggled PP1-9 are left executing IAM instructions on their respective channels. It is necessary to free these channels for use in I/O and to do so in such a manner that the PP's are not lost forever. The basic technique is to pass a small program to each PP which scans a location in CM for some signal and inputs further information upon receipt of the appropriate signal.

This process proceeds as follows: Upon receipt of a signal, the receiving PP activates the channel and executes an IAM. (The hardware sets up the PP's in this condition initially.) The sending PP detects an active channel, indicating that the other PP is ready, and executes an OAM on the channel. When all data has been output, the sending PP disconnects the channel, causing the receiving PP to exit from the IAM and begin executing the new code.

The first application of this technique is the freeing of channels at the beginning of CED by the subroutine FRECHAN. This is necessary because at Dead-Start time all PP's are connected to their corresponding channels and therefore PP0 is not free to use them. For example, suppose the display console is on channel 8 and PP0 wants to display a message. The problem is that PP8 is trying to read on channel 8 and if PP0 tries to use it the PP's will hang each other. To keep this from happening, FRECHAN sends a little idle loop to each PP on its corresponding channel and then disconnects the channel causing the PP to execute its idle loop (which just reads central memory) and then all channels are free for use by PP0.

The first thing FRECHAN does is to release the tape unit so that it can use the tape channel without interference from the tape unit. Before the next part is explained, the situation with the tape channel at Dead-Start time must be explained. If the tape channel is 12B or 13B then PP0 can read the tape without worrying about the other PP's since there are only 10 PP's and 12 channels, and channels 12B and 13B are not connected to any PP's. However, if the tape channel is 1--11B then Dead-Start must arrange it so that the corresponding PP reads some other channel when it read CEA and CED on the tape channel. The other channel it chooses is one of the free ones, 12B. For example, suppose the tape channel was 6. When the Dead-Start button is pushed, PP6 begins to read channel 6. Now, PP0 cannot read CEA from the tape on channel 6 because it would hang up with PP6. Therefore, che Dead-Start panel causes PP6 to start reading on channel 12B so that PP0 can read on channel 6. Now, with this background, the code between FRECHAN6 and FRECHAN2 can be detailed. It is necessary

to arrive at FRECHAN2 with each PP reading on its own channel. This will already be the case if the tape channel was 12B or 13B but will not be the case if the tape channel was 1-11B (remember if the tape channel was 6, the Dead-Start panel left PP6 reading channel 12B, not 6). Therefore, this code looks at the tape channel and if it is 1-11B, sends out the code from LITTLE TO ENDLIT on channel 12B to cause the PP to begin reading its own channel again. Therefore, FRECHAN2 is always arrived at with each PP reading its own channel and nobody reading channels 12B or 13B. Channels 12B and 13B are then disconnected if they are not already disconnected.

Channels 1-11B should be disconnected but not without first sending out a little program to each PP to execute, so it is signalled to begin input again when it is required to do another job (such as later accept PP-resident). However, since it may be desirable later to dump these PP's, the little program cannot be too big or it will erase information the operator might want to dump. The smallest program which can be sent them is one to read in one CM word.

The CM words 7767B - 7777B are chosen as the words that the PP's will read. PP1 will read 7767B, PP2 will read 7770B, etc. These words will contain the idle program so that each PP can read the program right in over itself and continue to execute it until a change is made to the program in the CM word. Therefore, the original contents of 7767B - 7777B are saved and the following little five byte program is put in each:

```
LDD  0
CRD  1
SBN  1
NJN  *-1
UJN  *-4
```

Then the following program is sent over each channel to each PP;

```
LDC  XXXX
STD  0
CRD  1
UJN  *-4
```

where XXXX = 7767B for PP1 and 7770B for PP2, etc.

The program sent over the channel will cause the address of the appropriate CM word to be stored in location 0 in the PP which will later be used to tell the PP which word to read. It is also used as a delay count in reading CM so that the PP's do not clog the read pyramid. This program will be sent to each PP over its corresponding channel and then the channel will be disconnected, causing the PP to execute the program and read its corresponding CM word. This will in turn cause the PP's to continue reading the CM word. When it is time for the PP's to begin inputting again, the contents of the CM word are changed to an input instruction. Also each channel is disconnected as the little program is sent out to each PP, meaning that all channels are now disconnected and PP0 can use any of them it wishes to read tape and display on the console. Thus, FRECHAN has now freed up all of the channels.

After determining that no dump is required, CED executes BEGINPT. This subroutine is used to cause PP1 - PP8 to exit from the little idle loops described in the discussion of subroutine FRECHAN and to cause these PP's to begin inputting on their own channel in the same manner that they were doing at the beginning of CED by virtue of the Dead-Start hardware. Later, SENLOOP will send out a bigger idle loop to each PP. After putting each new sequence of code in the flag cells (7767B - 7777B), BEGINPT waits until each channel has gone active (meaning that each PP has accepted its code from the flag cells and is now inputting on its own channel), and then restores the original contents of the flag cells which were saved at TABTCH by the subroutine FRECHAN.

Remember in the discussion of FRECHAN, it was stated that the idle loop must be small so as not to overlay cells in case of a dump. Since the dump has not been selected, it is now desirable to provide a bigger loop that will only look at cell 0 in CM and free the other cells (7767B - 7777B) for use for other things. Therefore, SENLOOP sends out the loop which appears in LOOP through ENDLOOP to each PP. This loop is bigger but now it does not matter if a few more cells in each PP are wiped out. The loop causes each PP to look for its number in CM location 0.

From this point CM location 0 is used to signal PP's to input on channel 0. This technique is used in the following cases:

1. By CED to send P or IRP to PP1.

2. By IRP to send RMS control to PP2.

3. By STL to send PPRES to PP2-8, MTR to PP0, and DSD to PP9.

4.8.0 OPCOM--The Operator Communication Package

Operator communications at deadstart are handled by several programs and a set of macros. A general characteristic of the communication's package is the uniform interface which it presents to the user in a PP or a CP. The same macros may be used to perform communications in both the PP and the CP. The macros and routines take care of the various conditions and situations which may occur.

There are three logical parts to the communication package:

- the macros which generate messages and calls to the display routines;
- the OPCOM routine (one version for the PP, one for the CP) which builds the hardware display;
- the display driver package which takes the output from OPCOM and displays it on the console CRT.

We will explain each of these parts in turn.

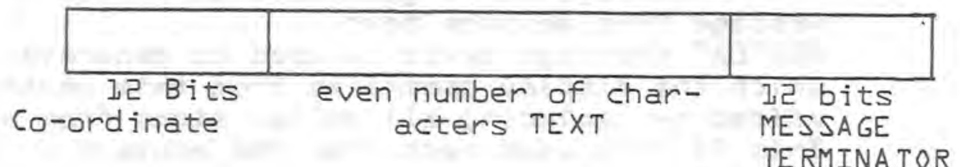
4.8.1 The Macros

The macros fall into three groups:

- the message generation macros;
- the message delimitor macros;
- the OPCOM call macro.

4.8.1.1 Message Generation

The message generator macros are used to generate co-ordinates and text for the display. The general form of a message is:



The co-ordinate is a twelve bit quantity which defines the position on the screen of the first character. The first six bits are treated as the Y-co-ordinate and the second six bits are treated as the X-co-ordinate. These co-ordinates are not the physical co-ordinates which are sent to the console. They are for the Y-co-ordinate, the number of lines down from the top of the screen and for the X-co-ordinate, the number of characters over from the

left side of the screen. These co-ordinates are translated into the actual screen co-ordinates by OPCOM. The screen co-ordinate generated depends upon the character size in effect at the time that the message is processed (see the message delimitor macros).

The text is any number of non-zero bytes of character data which has no character of display code value of 60B or greater present. The text is processed a byte at a time by the OPCOM package. Depending upon the type of message (explained below), the OPCOM package will examine each byte of text for a pair of dollar signs ($$) which serve as flags or special delimitors. A single dollar sign in a byte will be ignored. A pair of dollar signs which occur, the first in the last character of a byte and the second in the first character of the next byte will be ignored. Only a byte of dollar sign will be meaningful as delimitors. For the convenience of the programmer two modes of text generation are provided--the scanned mode and the unscanned mode.

The unscanned mode generates text which is identical to the message as it appears in the macro call. A single dollar sign or pair of dollar signs occuring in ajoining bytes will not be altered. In the scanned mode each pair of dollar signs is adjusted to occur in a byte if it does not by inserting a blank into the generated message before the dollar signs. A single dollar sign is replaced by two dollar signs and handled as above. The scanned mode requires considerable more assembly time. Accordingly, the programmer may turn the scan off and on by the use of the SMODE macro (see delimitor macros).

The message terminator is one or more bytes of zero.

The following macros generate messages:
TEXT (message text) generates a message identical to <u>message text</u>. The text is not scanned.
STEXT (message text) generates an edited or scanned message from message text.
MSGTEXT (message text) is used to generate a message in which the display generated from this message has been edited by replacing all dollar signs from a fill text area. This fill is used each time the message is generated and simplifies the insertion of small amounts of variable text data into the message.
QSTEXT (message text) is used to generate questions. Dollar signs are not replaced from fill text.
OPTEXT (message text) is used to generate options for the operator. The first set of dollar signs in the message delimits the end of the option name which must be ten characters or less. The text which follows the first set of dollar signs is optional and would generally be used to

explain the option to the operator. The entire message is displayed. {For a more complete explanation of options see the OPCOM macro.}

At the end of each message the X-co-ordinate is set to the left margin {the value of the SET symbol .$TABX} and the Y co-ordinate is upped by one to create a new line. The next message generated will use these co-ordinates, unless they are changed with positioning macros.

If the message text area is replaced by a name prefixed with a single dollar sign, the OPCOM package will use the co-ordinates of this message but the text from the message found at name. This is a simple form of indirect addressing.

4.8.1.2 Message Delimitors

The message delimitor macros are used to control the positioning, scanning, screen selection and character size of messages generated by the text macros.

SMODE X is the scanning control macro. If X is ▽ON▽, the scanning is turned on; if X is ▽OFF▽, the scanning is turned off. Any other value turns the scanning off. The scanning mode is off until turned on.

TABX X controls the positioning of the text on a single line. If X is numeric, the following text is to start at the **Xth** character of the line. If X is numeric preceded by a dollar sign, the text is to start at the Xth character of the last line. {Remember that the line number is advanced after each message. To tab on the line last generated requires the dollar sign form.}

PTABX X sets the left margin for all following text. If X is ▽⋈▽, zero is assumed.

POSTXT X, Y set the text position to X and Y. If X is ▽⋈▽, it is set to zero. If Y is ▽⋈▽, the current Y position is used.

The screen selection and character size are specified for a group of messages which are headed by a list header macro and terminated by a bit end macro. Besides the screen and character size selection, each group of messages also is a message group, a question group or an option group. Each group must be of only one type of message. The following are the list header macros:

XXXLST a,b is the general form of the macros. The parameters a and b are SMALL, MEDIUM, LARGE, RIGHT

or LEFT in any order or combination. The XXX is OPT for an option list, MSG for a message bit and QST for a question list.

ENDLST terminates a particular list (of any type). Only messages and delimitor macros (other than XXXLST and ENDLST) should appear between a list header and its matching list terminator.

For each type of list, there are initial position, character size, and screen selection assumptions. For all list types the character size is assumed medium. For MSGLST the screen selection is right, for QSTLST and OPTLST the screen selection is left. For MSGLST the initial position is the top of the screen. For OPTLST it is in the middle of the screen. For QSTLST it is one third down the screen.

4.8.1.3 OPCOM Call Macro

The OPCOM call macro controls what message groups are to be displayed, the type of operator input and the address of data area to be used to replace dollar signs in MSGTEXT. The form of the call is:
name OPCOM return type, group list, fill parameter
where name is optional.

Return type defines what type of input is expected from the operator. The following types are defined:

A requests that the OPCOM package returns the first word address of keyboard input and the length. In the PP the length is returned as a byte count in a direct cell called BLTH. The address is returned in the A register. In the CP the length is returned in X0 and the address in A1. The input buffer is six words (60 characters) in length and is blank (55B) filled.

N requests that the operator octal input be converted to binary and the value returned. In the PP the value is returned in the A register. In the CP the value is returned in X1. The length is returned as in A. If the input was not numeric, a negative value is returned.

O requests that the OPCOM package return the ordinal of the option specified by the operator. The operator must type in one of the option names specified by all OPTLST's and OPTEXT's or a carriage return only. Each option is numbered, from one, as it is processed. The carriage return alone is always option zero. When the operator types something and terminates it

with a carriage return, the input is matched to the list of options. If it does not match, the input is displayed again with an error message. If it does, the ordinal is returned in the A register for the PP or X1 for the CP.

In the above cases, the display remains up until the operator finishes keying in input. In the X display, the display remains up until the caller replaces it with a new display. The operator input is expected, required or allowed.

The group list specifies what message lists are to be displayed for this call. The form of the list is:
{a, b, ..., Z} where a, b, ..., Z are each the address of a message list header.

The fill parameter consists of a fill address and a fill byte count in the following form:
{fill address, byte count}.

A maximum of byte count bytes are taken from the area starting with fill address which is considered to be a continuous stream of bytes. If more bytes than byte count are required, blanks are used. These bytes are used to replace dollar signs in MSGTEXT messages.

4.8.2 The OPCOM Routine

There are two OPCOM routines--one for the CP and one for the PP. To the programmer both routines are identical with respect to input and the call. We will refer to these two routines collectively as OPCOM. The primary purpose of OPCOM is to generate a display buffer in a form suitable for a display driver. It does not do the actual displaying. The functions of OPCOM are:

- Generation of console display function byte
- Translation of line and character co-ordinates to physical co-ordinates
- Replacement of dollar sign bytes in MSGTEXT messages
- Determination of allowable options
- Transmission of buffer to display driver
- Analysis of input if necessary to determine what option was selected
- Conversion of numeric input

The PP OPCOM is also responsible for the determination of where the display driver is. If the call for the display is made in PP zero, the display buffer is built in the high addresses of PP zero and the display driver is called directly. If the call is from any other PP or from the

CP, the display buffer is built in CM and a controlling program in PP zero reads the buffer to PP zero and calls the display drivers.

4.8.3 The Display Driver

The display driver is divided into three parts--the keyboard monitor, the CRT driver and the controlling routine which calls the first two and handles the communication with other PP's and the CP.

The keyboard monitor reads in the characters typed by the operator. If they are control characters such as backspace or blankout, it takes the appropriate action. If they are text characters, it assembles these characters into a buffer which is displayed at the bottom of the left screen. If the control character is a carriage return, it returns a flag to the caller. The keyboard monitor also takes care of single word displays and alterations to memory. If the operator types six or less octal digits followed by an equal sign at the beginning of the buffer, the keyboard monitor will display that word of central memory with the address specified by the six or less octal digits. If the operator follows this by one or more octal digits (which may be separated by non octal digits), the keyboard monitor will store the right most twenty or less octal digits into the word left filled with binary zeros.

The display driver is called to write a display buffer to the console CRT. The buffer consists of oneor more message groups of the following form:

Function	Co-ordinates and Text	Terminator

The function specifies screen and character size. It is sent directly to the console CRT.

The co-ordinates and text are sent to the console CRT as data until the terminator is encountered. The terminator is one or more bytes of zero which signal the driver that a new function follows the last zero byte of the terminator. If the function is all ones, the driver returns to the caller.

The controlling routine calls the driver and keyboard monitor to drive the display. It is also responsible for

communications with other processors. It receives a flag word which contains:

- a display flag which is on when the display is on;
- an input flag which signals that input from the operator is to be passed back to the calling processor;
- a byte count of input to be redisplayed upon an error. This is used to return the byte count of input to the caller;
- a buffer flag which is on if the display driver has not read the buffer and off if it has read the buffer.
- an error flag.

When the operator completes his input, the display and input flags are turned off and the control word in CM is reset.

There is an alternate controlling routine when the display call is made in PP zero.

4.9.0 Dead-Start Debugging Aids

The following debugging aids are available for use with the dead-start routines:

1. One-word central memory display
2. IRCP central processor breakpoint routine
3. Device read/write routine

4.9.1 One-word Central Memory Display

Throughout the time in which the display driver is executing in PP0, it is possible to examine and/or change the contents of any single location in central memory. The manner in which this works is shown by the following example.

The operator types:

a =

Where a is 1-6 octal digits designating a CM address. The display driver recognizes this as a special form of type-in, even when other operator input is being received (octal digits followed immediately by the equal sign). Once this form is recognized, PP0 displays the contents of this location as follows at the bottom of the left screen:

a = c

Where c is 20 octal digits. This value is updated dynamically as the contents of this word changes. If the operator now types a CR, the a = portion of the display disappears, but the contents remain, and thus other type-ins may be made while still observing the contents of the word. If instead, the operator wishes to change the contents of this word, he may, after entering the = sign, enter from 1 to 20 octal digits. This is displayed as follows:

```
          c
a   =     n
```

Where c is the same 20 octal digits as before, representing the actual contents of the word, and n is 1-20 octal digits entered by the operator. If the operator now types a CR, the value represented by n is right-justified, and is then stored in CM. Now, only the top line remains on the screen, but it shows the contents of the word after the storing takes place. Again, if the carriage return is typed immediately after the equal sign, no storing takes place.

The blankout key and backspace key can both be used for their normal functions.

4.9.2 IRCP Central Processor Breakpoint Routine

This routine can be used for console debugging of IRCP by setting breakpoints, examining the contents of the registers and of central memory, and by changing the contents of central memory. Its use is described in detail in the listing of IRCP.

The code for the breakpoint routine is assembled only if the value of the symbol BREAKPT is non-zero. The symbol is defined in DSLCOM. Most of the code is in IRCP and some is in CONTROL. The computer must have the central exchange jump instruction (XJ) in order that the breakpoint routine can be used. CONTROL, STL, and IRCP must be reassembled.

4.9.3 Device Read/Write Roution

A routine called DEBUG is available in IRCP to read (dump) and write selected portions of any device. It is assembled only if the DSLCOM symbol XOPTION has a non-zero value

The routine is entered by selecting a dead-start option of 1.X when the large display of all the dead-start options appears. This causes CED to give control to IRCP with

everything set up in the same way as if an initial, normal, or recovery dead-start were being performed. IRCP transfers immediately to DEBUG. None of the usual preliminary functions such as label processing are performed.

The operator entries are apparent as the routine is used. The operator is first asked to enter the EST ordinal of the device to be referenced. Once entered, this can not be changed without dead-starting again. Then one of the following features of the routine is selected. This is also permanent, once selected:

1 - Erase one RB
2 - Dump one RB
3 - Dump one PRU

After one of the above is selected, the main loop of the routine is entered. Each option is processed as follows:

Option 1: The operator is asked to enter an RB number Then a buffer the length of an entire RB is filled with a preset value. The buffer is then written to the selected RB by using RMS control. This process is then repeated until the operator dead-starts again.

Option 2: The operator is asked to enter an RB number. RMS is then called to read the RB into CM. The input area is designated as starting at RMSBUF1 (42500B). This process may be repeated as many times as the operator wishes or until CM is exhausted, for which no checking takes place. Each successive RB selected is read starting at the LWA+1 of the previous read rounded up to the next 100B. The area between reads is filled with all ones for easy observation. Also, to aid in identification, the word immediately preceeding each read is set with the following:

Bits 0-11 = PRU number
Bits 12-23 = RB number
Bits 24-59 = Zero

As should be apparent by now, it is necessary to use dead-start dump to get this output. After each read is completed, a message is displayed which indicates where the last read began.

Option 3: This operates in the same manner as Option 2 except for the following differences:

1. The operator is asked to enter both an RB number and then a PRU number.

2. The header information described above appears at the start of each PRU, whereas, in option 2 it appears only at the start of each RB. For this same reason, in option 2, the PRU number will always be equal to zero.

3. CM is used up much less rapidly, since only one PRU at a time is placed in CM. Actually, due to the header information, the PRU header bytes, and the rounding up as described above, each PRU dump uses a total of 200B CM words.

CHAPTER 5 - TABLE OF CONTENTS

5.0 MTR - SCOPE SYSTEM MONITOR 5-1

5.1 BASIC DEFINITIONS 5-1

5.2 MAIN LOOP AND MAIN CONTROL 5-4

- 5.2.1 Output Register Scanning 5-5
- 5.2.2 Advance Clock 5-5
- 5.2.3 Advance Control Point 5-6
- 5.2.4 Process Delay Stack 5-6
- 5.2.5 Recompute Priority Sublevel 5-6

5.3 CPU REQUEST PROCESSING 5-7

5.4 PPU REQUEST PROCESSING 5-9

5.5 MEMORY ALLOCATION - STORAGE MOVE 5-21

5.6 CPU ASSIGNMENT 5-22

5.7 PPU ASSIGNMENT 5-23

- 5.7.1 PPU Status Table 5-23
- 5.7.2 PP Queue 5-24

5.0 MTR - SCOPE SYSTEM MONITOR

MTR is loaded into PPU zero at dead-start time and remains there for the duration of system execution.

The primary task of MTR is to control and coordinate system activities in order to avoid any conflicts between the various processors.

MTR will for instance

- control the execution of CPU programs
- assign jobs to pool peripheral processors
- allocate central memory storage (and eventually ECS) to various control points
- reserve and assign data channels and equipment
- periodically evaluate control point priorities
- maintain the system clock and accounting

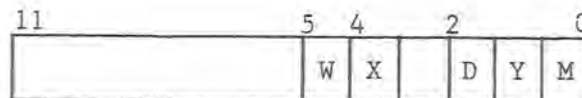
All of MTR's tasks are sensed during its course through the MTR main loop. From there, five different modules can be entered; the chart on page 5-2 shows this module organization and indicates in which section of this chapter the module description can be found.

5.1 BASIC DEFINITIONS

Control Point Status

Byte C.CPSTAT of word W.CPSTAT in the control point area contains the status of assignment of the central processor to the control point.

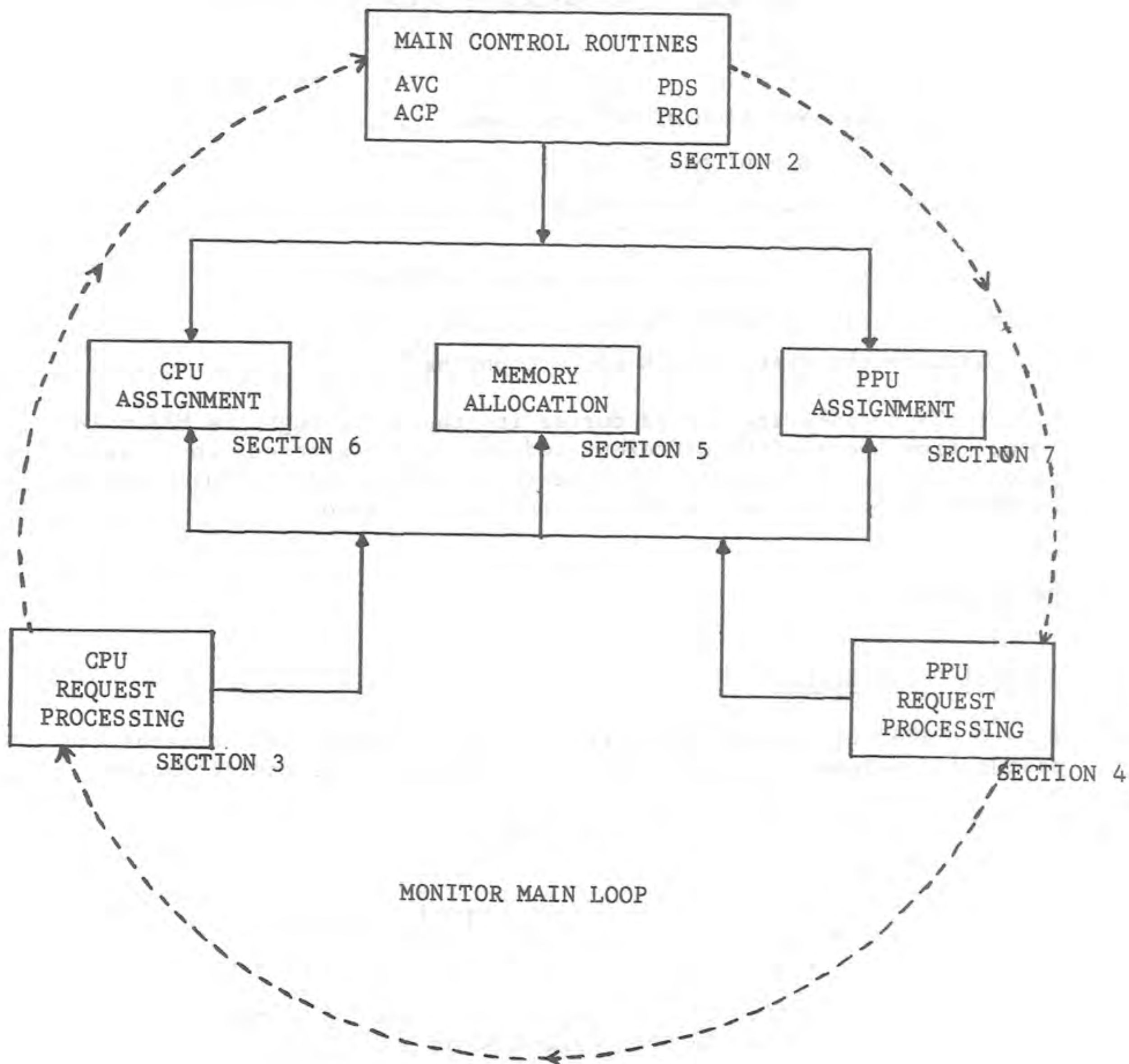
The format of this byte is the following:



- bits 6 to 11 are used internally by MTR for the CPU assignment in a multiple CPU configuration
- bits 4 and 5 are the control point status bits
- bits 0 to 3 are the secondary status bits

At any given time, a control point is in one of the following three statuses:

1. Zero status or Z status (Status bits: W=X=0)
   A control point is in Z status if it has no requirements for the CPU.


MAIN CONTROL ROUTINES
AVC
ACP
PDS
PRC
SECTION 2
CPU ASSIGNMENT
SECTION 6
MEMORY ALLOCATION
SECTION 5
PPU ASSIGNMENT
SECTION 7
CPU REQUEST PROCESSING
SECTION 3
PPU REQUEST PROCESSING
SECTION 4
MONITOR MAIN LOOP

2. Recall status or X status (Status bits: W=0, X=1)
   A control point is in the recall status if it has relinquished the central processor while waiting for a peripheral task to be completed.

3. Waiting status or W status (Status bits: W=1, X=0)
   A control point is in the waiting status if it currently has need for the CPU.

   A control point in the W status is said to be active or running if it has currently control of a central processor. The control point area address of an active control point is recorded in the CMR pointer area, word T.STATCP, byte 3 for CPUB, byte 4 for CPUA. (If zero, the processor is idling).

Secondary status bits:

- A control point is said delegated when it can only become active in a specific CPU and no other program can receive control of this CPU as long as the delegation persists. In this case the secondary status bit, D, is set in the control point status.
- A control point in the X status is said to be in periodic recall if it has relinquished the CPU for a certain time period. It is said in automatic recall if it has relinquished the CPU until a certain status is set complete by some peripheral task.
  A control point in automatic recall is characterized by the secondary status bit Y set in its status byte.
- MTR may decide to suspend temporarily a control point in order to move its storage. In this case, the secondary status bit M will be set.

CPU Status

Monitor maintains in PP0 memory a 12-bit status for the central processor(s).

This status can be:

7777	if this processor does not exist (6500 MTR running on a 6400)
0060	if this processor is turned off
0040	if the processor is on
0044	if this processor is delegated to a specific control point.

In the last two cases, the central processor status is identical to the status of the control points eligible for assignment of this CPU.

In a single processor machine, the CPU status can only be 0040 (W status).

The status of the CPU(s) is recorded in the CMR pointer area, word T.STATCP, byte C.STATCP. This byte contains the 6 right bits of both CPU(s) status in the form:

CPU B status	CPU A status

Example: For a 6600 or 6400 the contents of C.STATCP is 7740.
For a 6500 with CPU B delegated the contents is 4440.

Control Point Activity

Three different activity counts are maintained by MTR for every control point:

1. The General Activity Count which contains 1
   - for each PPU assigned
   - for each outstanding request for a PP job, with or without time delay
   - for each uncompleted stack request (at this control point)

2. The Pseudo-Activity Count which can only be incremented or decremented via the MTR function M.RACT.

3. The PP Delay Count which contains 1 for each outstanding PP job requested whose time delay has not yet expired.

Monitor considers that there is no activity at a control point if the sum of the General Activity Count and the Pseudo-Activity Count is zero.

The PP Delay Count is only maintained to allow a PP routine like 1RO to differentiate outstanding requests with delay from the others.

Note:
All the activity counts are maintained in PP0 memory; they are not stored or updated in central memory.

5.2 MAIN LOOP AND MAIN CONTROL

In its main loop, Monitor

- scans the output registers of the assigned PPU's

- scans the relative location one {RA+1} of the active control point{s}
- calls the following four control routines:
  AVC - Advance Clock
  ACP - Advance Control Point
  PDS - Process Delay Stack
  PRC - Priority Recomputation

The sequence of operations is the following:

1. Test if PPU is assigned and a request is in its output register. If not jump to step 3.

2. Process the PPU request {PPM}
   Advance the clock {AVC}
   Process eventual CPU request{s} {AVCPU}

3. If all PPU's have not been processed, go back to step 1 for the next PPU.

4. Process delay stack {PDS}
   Process CPU request{s} {AVCPU}
   Recompute priority sublevels {PRC}
   Advance the clock {AVC}
   Advance a control point {ACP}

5. End of the loop. Jump back to step 1 and start with PPU1

5.2.1 <u>Output Register Scanning</u>

In the main loop the code necessary to scan a PPU output register is repeated for each PPU.

Each sequence is eight words long which is the same length as the PP communication area in central memory. This equality of length permits addressing of words within a sequence for a specific PPU, by using the output register address of this PPU.

There are two words within the sequence which are frequently modified:

* the first instruction is
  UJN *+8 if the PP is not assigned
  LDC   if the PP is assigned
* the last instruction LJM PPM is modified if a PPU request processing routine makes a direct return jump to the main loop.

5.2.2 <u>Advance Clock</u>

This routine must be entered at least every 4 milliseconds.

Therefore it is entered after the processing of any PPU or CPU request.

Its function is to maintain the real time clock and the display code clock stored in the CMR pointer area respectively at T.MSC and T.CLK.

5.2.3 Advance Control Point

This routine is entered once every pass through the main loop. It analyzes the status and the activity of the control points {one control point every time} and will:

- call 1AJ at a control point if no activity remains at this control point
- decrement the recall delay if the control point is in periodic recall {X status}
- set the error flag F.ERHANG if no activity remains at a control point in automatic recall {X+Y status}
- set the error flag F.ERTL if the time limit flag is set at the control point.

5.2.4 Process Delay Stack

This routine is entered once every pass through the main loop. It scans the queue of PP job requests issued with a time delay and initiates the peripheral jobs whose delay has expired.

5.2.5 Recompute Priority Sublevel

Every 2** {IP.CPD+6} milliseconds, MTR re-evaluates the priority sublevel of the control points who have a non-fixed or non-zero priority.

An installation may specify several parameters for the scheduling algorithm used in computing priorities.

a} IP.MPR designates the maximum priority level which may be assigned to a job. This is the largest number which a user may specify as priority on his job card. Possible values of IP.MPR range from 0 to 7777_8. The value of IP.MPR determines the number of bits used for the level {highorder portion of priority}.

b} IP.LVF indicates the lowest fixed priority level. The sublevel of any priority whose level is greater than or equal to IP.LFV will not be recomputed.

c) IP.CPD, as mentioned previously, defines the interval between priority recomputation of jobs at control points. (Jobs with a fixed priority level are not affected by this recomputation.)

d) IP.OSW is a weight factor applied to the old priority sublevel during priority recomputation. The weight used is $2^{IP.OSW-1}$; possible values range from 0 to 6. Thus, the rate at which a priority sublevel changes when the character of the job changes, may be controlled.

To re-evaluate sublevels, MTR maintains at each control point the cumulated milliseconds of CPU time tallied at the control point. This sum is reset to zero each time a priority level or sublevel change occurs. If CPT is this sum and:

T = 2**(IP.CPD+6)
W = 2**IP.OSW
N = Number of bits in the sublevel
A = 2**(IP.CPD+6-N)
S_i = old sublevel

then the new sublevel becomes:

$$S = \frac{\frac{T-CPT}{A} + (W-1)*S_i}{W}$$

5.3 CPU REQUEST PROCESSING

MTR distinguishes two types of central programs:

- System programs which run at control point N.CP+1 with RA = 0, FL = 377777B and a priority of 7777. These programs cannot make requests through RA+1. They always terminate with a jump to zero (location counter P = 0)
- Problem programs which run at control point 1 through N.CP. These programs communicate with MTR via RA+1: MTR scans periodically the contents of word RA+1 of the active program(s). If it becomes non-zero, MTR will consider it as a CPU request and process it before clearing RA+1.

The first three characters of RA+1 give the request type: if

END - End central program -
the control point is put in the Zero status

ABT - Abort control point -
the error flag is set to F.ERCP at the control point and the control point is put in the Zero status.

RCL - Enter recall status -
1-if the automatic recall bit is not set, the control point is put in the X status and a periodic recall delay is initialized
2-if the automatic recall bit {bit 40} is set and the parameter address {bits 0-16} points to an incomplete status {bit 0 not set}, the control point is put in the X+Y status
3-if the automatic recall bit is set and the parameter address points to a complete status, the request is considered as a no-operation.

TIM - Supply time or date -
Depending on the value of the parameter transmitted in RA+1 {bits 24-35}, MTR will return at the location specified in bits 0-16 the following information:

parameter value		
	0	elapsed CPU time {W.CPTIME}
	1	{T.DATE} see CMR pointer area
	2	{T.CLK} " " " "
	3	{T.JDATE} " " " "
	4	{T.MSC} " " " "

if else - MTR assumes it is a PP program call
In this case, the first character must be alphabetic. MTR will take the contents of RA+1, clear bits 36-39 and 41, insert the control number, consider this as a PPU input register image and assign a PPU. If the automatic recall bit is set and the parameter address points to an incomplete status, the control point is put in the X+Y status.
If the automatic recall bit is set and the parameter address points to a status already complete, the control point is aborted {error flag F.ERRCL}

During the process of central processor requests, the following error flags can be set:

F.ERAR - Arithmetic error
Error detected when the location counter P is zero for a problem program

F.ERCP - Control point aborted
Error flag set in answer of a ABT request

F.ERPCE - PP call error
Error detected when a PP program call does not begin with an alphabetical character or when an invalid parameter is transmitted {address not in the control point FL for instance}

F.ERRCL - Automatic recall error
Error detected when a PP program call is issued with the automatic recall bit set and the parameter address transmitted points to a status already complete.

When a valid request with auto-recall bit and a status address is issued, MTR sets the control point in the X+Y status. Later on, every time the general activity count of this control point is reduced, MTR will check the status and put the control point in the W status if the completion bit is set.

For this purpose when such a request is detected in RA+1, MTR records the status address by writing the complete request from RA+1 into word W.CPAR of the control point area.

5.4 PPU REQUEST PROCESSING

All PPU requests are made by placing the number of the request (M.xxxx) in byte zero of the PPU output register. Parameters for the various requests are supplied to MTR via bytes 1-4 of the output register and/or the PPU message buffer. Upon completion of the request, MTR replies by setting byte 0 of the output register to zero and gives the response back to the requesting PPU in the remaining bytes of the output register or the message buffer.

If MTR detects that the format of the request is bad, it sets the high order bit of byte 0 of the appropriate PPU output register. This hangs the PPU, because MTR will ignore any request with this bit set. An appropriate message: PPn NAMcp BAD MTR REQUEST is inserted in MTR's message buffer and will be displayed at the bottom of the right screen in flashing characters.

The following table lists all current MTR function codes and the associated mnemonics.

01	M.DFM
02	M.RCH
03	Reserved for CDC
04	M.PPTIME
05	M.STEP
06	M.ICE
07	M.RBTSTO
10	M.RSTOR
11	Reserved for CDC
12	M.DPP
13	M.ABORT
14	M.NTIME
15	M.RCP
16	M.DCP
17	M.PAUSE
20	M.RPP
21	M.RCLCP
22	M.REQP
23	M.DEQP
24	M.RPRI
25	M.REM
26	M.SEQ
27	M.RACT
30	M.SEF
31	M.RTAPE
32	M.DTAPE
33	M.ESTZ
34	M.EREQS
35	M.CCPA
36	M.CPUST
37	M.RPJ
40-47	Reserved for installations
50-77	Reserved for CDC

In the following pages, a complete description of the contents of the output register and the function parameters is given for each request. Those bits or bytes which are irrelevant to the function are denoted by *'s.

(The requests are listed in the alphabetical order of their SCPTEXT mnemonics.)

M.ABORT - ABORT CONTROL POINT

(0013,****,****,****,****)

The job associated with the requesting PPU is terminated. The requesting processor is responsible for an explanatory message in the dayfile.

The operation of this function is identical with function M.DPP except that the error flag in the control point area is set to F.ERPP(3) to note the abort function.

M.CCPA - CHANGE CONTROL POINT ASSIGNMENT

(0035,****,****,***C,***N)

The requesting PPU is released from its current control point assignment in the same manner as if it had issued an M.DPP function, but its input register is not cleared. The PPU is then assigned to control point N with the new control point number inserted in its input register.

If C is non-zero, the activity count of control point C will be reduced by one (this option should only by used by stack processor).

M.CPUST - CHANGE CPU STATUS (6500 ONLY)

(0036,***X,****,****,***N)

Option 1: N is non-zero.
The request is ignored if a CPU is OFF or delegated. If this is not the case, CPU X (X = 1 or 2) is delegated to control point number N. No other control point may use a CPU delegated to control point N.

Option 2: N = 0, X = 0
If either CPU is off or delegated, it is returned to the ON status. This does not affect a CPU that was locked off at deadstart load time.

Option 3: N = 0, X = 1 or 2
CPU X is turned off. If one of the CPUs was delegated, the delegation is cancelled.

M.DCP - DROP CENTRAL PROGRAM

(0016****,****,****,****)

Execution of the central processor job at control point is stopped. The control point status is set to Z (zero status), the secondary status is not altered.

The control point status bits prior to M.DCP are returned in byte 1 of the output register of the requesting PPU.

M.DEQP - DROP EQUIPMENT

(0023,00EE,****,****,***N)

MTR drops equipment ordinal EE from the control point and updates the EST entry to indicate that this equipment is free for reassignment. There is no check by MTR to ensure that the dropped equipment was assigned to this control point. The parameter N gives the control point number to be considered if the requesting PP is attached to control point zero, otherwise it is irrevelant.

M.DFM - PROCESS DAYFILE MESSAGE

(0001,FFFF,MMMM,****,****)

The dayfile flag bits FFFF determine (when set) the following message handling:

bit 0	Do not send to B display
bit 1	Do not send to the control point dayfile
bit 2	Do not send to system dayfile (No A display)
bit 3	Flag as an accounting message
bit 4	Send to hardware error file
bit 5	Do not insert the job name in the system dayfile

The parameter MMMM gives the last word address of the message in the PP message buffer or gives a dump index when a dayfile dump is requested.

The possible value of the dayfile dump index is:

0	for a system dayfile dump
1 through N.CP	for a control point dayfile dump
N.CP+1	for a hardware error file dump

M.DPP - DROP PP

(0012,****,****,****,****)

MTR clears the PP control point assignment (the PP status word and the PP input register are cleared).

M.DTAPE - TURN EQUIPMENT OFF

(0032,00EE,****,****,****)

MTR will turn the equipment EE logically off by setting the on/off lockout bit in the relevant EST entry.

M.EREQS - ENTER REQUEST STACK

(0034,****,00AA,SSSS,00BB)

((C.RQSFS)+SSSS)*2 is the address of a CM word pair in the common request stack. The requesting PP has the responsibility to search for an empty entry in the stack and to communicate this word pair location in the SSSS field.

If MTR detects that the specified word pair is already in use, it will refuse the request by setting byte 1 of the output register to 0001 and clearing byte 0.

If the word pair is available, the stack request is taken from the PP message buffer and stored in the available word pair. The entry count of the appropriate device status entry and the control point activity count are increased by one. Finally, the entire output register of the requesting PP is set to zero to indicate that the request has been completed.

If AA = 0, control is returned to the PP normally.

If AA = 77B the request will be reissued for the device BB, i.e., MTR will update the appropriate device status table and control point activity counts, without inserting a new entry in the common request stack. (If AA is not 77, the last byte of the output register is ignored)

If AA is not 0 or 77, the requesting PP is available for reassignment. It is released in the same manner as if a M.DPP request were issued.

M.ESTZ - UPDATE EST Z BYTE

(0033,00EE,**ZZ,****,****)

This function alters the status field (bits 48-59 in the EST entry for any allocatable device. MTR makes the requested change only if it is legal in relation to the existing status.

EE is the EST ordinal of the device and ZZ is the right half of the status byte (bits 48-53) in the EST entry. MTR will respond to the request by copying the new status into bits 24-29 of the output register. Comparison with the request will show whether the request was granted.

The possible values of ZZ are:

00B = Public with no active files; at control point 0

10B = Public with one or more active files; at control point 0

20B = Private with no active files; at control point 0

2pB = Private with no active files; at control point $p \neq 0$

30B = Private with an active file; at control point 0

3pB = Private with an active file; at control point $p \neq 0$

40B = Logically unloaded; at control point 0

With the M.ESTZ function, only the changes indicated by X in table below can be made (the value of p cannot change, except to become zero).

from/to	00	10	20	2p	30	3p	40
00	X	X	X	X			X
10	X						
20	X		X	X	X	X	X
2p			X	X	X	X	X
30			X	X	X	X	X
3p			X	X	X	X	
40	X		X	X	X	X	X

M.ICE - INITIATE CENTRAL EXECUTIVE

(0006,****,****,****,IIII)

The parameter IIII identifies a central memory resident program which will be started by MTR upon recognition of this request. This system program will run at control point zero (RA = 0, FL = 377777B, priority = 7777B) and terminate by a jump to zero.

The M.ICE request is delayed if a system program is already active; MTR initiates only one system program at a time.

M.NTIME - ENTER NEW TIME LIMIT

(0014,TTTT,T****,****,***N)

A central processor job time limit of TTTTT seconds is entered at the control point. Any previous time limit is superseded.

If the requesting PPU is assigned to control point zero, the parameter N will give the Number of the control point to be considered; in any other case this parameter is irrelevant.

M.PAUSE- PAUSE FOR STORAGE RELOCATION

(0017,****,****,****,****)

This function tells the monitor that the PPU will not make any CM references within the field length of the associated control point until the function is completed. This permits monitor to move central storage for the control point; the function will not be completed as long as the storage move flag is set at the control point. The requesting PPU should check the reference address for the control point after completion of this function to determine if central storage for its job has been moved. (See PP Resident function R.PAUSE.)

M.PPTIME - ASSIGN PPU TIME

(0004,****,****,****,****)

MTR adds the current time minus the PPU starting time, to the accumulated PPU time in the control point area (word W.PPTIME).

M.RACT - REQUEST CONTROL POINT ACTIVITY

(0027,***N,IIII,****,****)

This request allows a PPU to know the various activity counts of control point N at a given time (N cannot be zero). If the parameter IIII is non-zero, the pseudo-activity count will be incremented or decremented by the constant IIII (after sign extension). The reply of monitor is made via the PPU output register:

Byte 1: control point status byte

Byte 2: general activity count

Byte 3: count of outstanding delayed PP requests

Byte 4: pseudo-activity count

M.RBTSTO - REDUCE RBT STORAGE

(0007,SSSS,****,****,****)

MTR sets SSSS*100B as the new RBT starting address.

M.RCH - REQUEST CHANNEL

(0002,BBAA,DDCC,****,RRRR)

AA = 1st choice channel number

BB = 2nd choice channel number

CC = 3rd choice channel number

DD = 4th choice channel number

RRRR = 0000 Request immediate reply

RRRR ≠ 0000 No reply until a requested channel has been reserved.

When channel zero is requested, it must be field AA. When BB, CC or DD is zero, it is assumed that this is not a channel request and that there are no alternate choices beyond it.

If none of the requested channels is available and an immediate reply is requested, MTR will set bytes 0 and 4 of the PPU output register to zero.

When a channel is granted, the number of that channel will be returned in the PPU output register byte 1 (location of AA). Byte 4 will be set to a non-zero value.

M.RCLCP - RECALL CENTRAL PROGRAM

(0021,****,****,****,****)

This request has only an effect if the central program associated with the requesting PPU is in the recall status and no error flag is set at the control point. In this case the status of the control point is set to waiting (W).

In any other case the status of the control point is not altered.

M.RCP - REQUEST CENTRAL PROCESSOR

(0015,****,****,****,****)

This request is ignored under the following conditions:

a. The requesting PPU is assigned to control point zero

b. The error flag is set for the control point

c. The job is already in the waiting status

If none of the above conditions exist, MTR will set the job in -

a. the automatic recall status (X+Y) if the auto-recall pointer still points to an incomplete status

b. the waiting status (W) in any other case.

M.REM - ASSIGN ERROR EXIT MODE

(0025,MMMM,****,****,****)

MTR assigns the value MMMM to the exit mode field in the control point exchange package area. (The control point cannot be in the waiting status.)

M.REQP - REQUEST EQUIPMENT

(0022,EEEE,****,****,***N)

The parameter EEEE consists of two display-coded characters:
if numeric it gives an EST ordinal,
if alphabetic it defines an equipment type.
MTR will search the Equipment Status Table (EST) to locate the appropriate EST entry. This entry is updated to reflect the assignment to the control point of the requesting PPU or to control point N if the PPU is assigned to control point zero. Finally, MTR places the assigned equipment ordinal in the first byte of the PPU message buffer.

If the equipment is not available, a zero byte is returned.

M.RPJ - REQUEST PERIPHERAL JOB

(0037,DDDD,DDDD,****,****)

This function requests that another PPU program be initiated after a specified time delay. The first word of the requesting PPU message buffer contains the input register image of the new PPU program. The time delay is DDDD DDDD*250 microseconds. (This time is rounded upward to the next larger millisecond.)
If the time delay is zero and no PPU is available, the request is entered in the PP job queue. If no space is available in the PP job queue buffer of MTR, the entire request remains pending until a queue entry becomes free.

M.RPP - REQUEST PPU

(0020,****,****,****,****)

This function requests immediate initiation of another PPU program. The first word of the requesting PPU message buffer contains the input register image of the new PPU including the control point number to which it should be assigned. The input register address of the assigned PPU is placed in the first byte of the requesting PPU message buffer. A zero byte is returned and the request is rejected if no PPU is currently available.

M.RPRI - REQUEST PRIORITY

(0024,PPPP,****,****,***N)

Assign priority PPPP to the control point of the requesting PPU if it is not control point zero. The parameter N gives the control point number to be considered if the requesting PPU is assigned to control point zero. In any other case the value of N is irrelevant.

M.RSTOR - REQUEST STORAGE

(0010,CCCC,XXXX,00TT,****)

Option 1 TT = 00 CM request only
= 01 ECS request only
= 02 CM and ECS request

Assign CCCC hundred octal words of central memory and/or XXXX thousand octal words of extended core storage to the control point of the requesting PPU.

Monitor replies to this request by setting CCCC and/or XXXX to the values actually assigned to the control point and by setting byte 0 to zero. These values should be compared with the original values requested to determine whether these requests have been honored or not. A request for more storage is rejected if not enough storage is available or if a storage move is already in progress. A request for less storage is always honored immediately. If TT = 02, MTR can honor a part of the request, without honoring the other.

Option 2 TT = 10 RBT storage request

CCCC*100B is the address of the lowest word in central memory requested by the stack processor as limit of the RBT storage area.

Monitor replies with the address actually allocated in CCCC.

M.RTAPE - TURN EQUIPMENT ON

(0031,00EE,****,****,****)

Monitor will clear the on/off lockout bit for equipment EE in the equipment status table (EST).

M.SEF - SET ERROR FLAG

(0030,***N,EEEE,****,****)

Monitor will drop the central program at control point N by putting the program in the zero status, and set the error flag to the value EEEE.

M.SEQ - ASSIGN JOB SEQUENCE NUMBER

(0026,****,****,****,****)

Monitor returns in byte 1 of the PPU output register a job sequence number (in display code).

M.STEP - MONITOR STEP CONTROL

(0005,****,****,****,***N)

This control is initiated by a keyboard request. MTR sets an internal step control flag and at each subsequent request MTR pauses for console keyboard input. A space from the keyboard causes MTR to process the request. A period from the keyboard causes MTR to process the request and clear the step control flag to resume high speed operation.

If N = 0 all PPU requests are stepped. If N is non-zero, control point N is the only one to be placed in step mode: only the requests issue by the PPU's assigned to control point N will be stepped.

5.5 MEMORY ALLOCATION - STORAGE MOVE

The storage associated with each control point is contiguous and assigned in the same sequence as the control point numbers.

Extended core storage is allocated in exactly the same way as central memory storage.

Low CM core is permanently assigned to control point zero (CMR).

High CM core is reserved for RBT storage.

The storage associated with each control point consists of two types, either of which may have a value zero:
a) allocated storage defined by the reference address and field length of the control point.
b) unallocated storage residing between the allocated portions of two consecutive control points. It is associated with the lower of the two control points but may be transferred to neighboring control points by moving any intervening allocated storage

Monitor maintains in PP0 memory two tables containing the sizes of allocated and unallocated storage (CM and ECS) for every control point (0 through N.CP)

A request for reduced field length will have the effect of transferring a portion of the storage from the allocated to the unallocated block; no storage is moved.

A request for an increased field length when the total storage associated with the control point is adequate, will result in a transfer of unallocated storage to allocated storage and will not cause any storage move to take place.

If it is necessary to take unallocated storage from the other control points, there will be a scan of control points above and below the requesting control point. This scan will locate the combination of blocks of unallocated storage which result in moving the least storage.

In any case, when a control point reference address and/or field length has to be modified, MTR will:
1. Suspend this control point by setting the M status and the storage move flag.
2. Wait for all PPU's assigned to this control point to pause (for CM move only)
3. If a move is necessary, initiate the system exchange package and start the storage move program.
4. After completion of the move, modify the control point reference address and/or field length
5. Resume the control point by clearing the M status.

Notes: 1. In case of a request for control point zero or a request for extended core storage, the storage move flag is not set at the control point and step 2 (Wait for pause) is bypassed.

2. A PPU is considered as pausing if its output register contains a M.RSTOR or M.PAUSE request or if it is hung with a bad MTR request (see 5.4)

3. If an ECS parity error is detected during an ECS move, MTR will abort the requesting control point and the control point being moved if it is not intact anymore. MTR will reject any further request for more ECS storage.

5.6 CPU ASSIGNMENT

To facilitate CPU assignment, MTR maintains in PP0 memory two tables with one entry per control point:

- the list of the status of all control points (0 through N.CP+1)
- the list of the priorities of the corresponding control points (priority 0 for control point zero, priority 7777 for N.CP+1)

At any given time, the CPU is allocated to the highest priority control point whose status matches the CPU status.

The CPU assignment routines are entered when:

A - A control point enters the W status or the priority of a control point in W status is modified. In this case MTR checks that the control point status matches a processor status and the control point will be initiated if its priority is higher or equals the priority of the active control point.

B - An active control point leaves the W status. In this case MTR searches the control point tables in order to find the highest priority control point whose status matches the status of the processor. This control point is then initiated (if it is control point zero, the idle program will be initiated).

A control point may enter the W status when it returns from recall status or when a M.RCP request is issued. MTR tries to bring a control point out of recall every time the general activity count of this control point is decremented, i.e., after a M.DPP or M.CCPA request. In this case, the control point is put in the W status if it was in periodic recall (x), or if it was in automatic recall (X+Y) and the recall pointer points to a complete status. In any case, a control point leaves the recall status if a M.RCLCP request is issued or if the time delay of a periodic recall is elapsed.

A control point may leave the W status after:

- a M.DCP, M.ABORT, M.SEF peripheral request
- a normal or abnormal termination of the central program (END or ABT in RA+1)
- a RCL or a PP program call with the automatic recall bit set, requested in RA+1.

Exchange package management
To transfer control form one control point to another, MTR executes a double exchange jump from the running program to the idle program, then from the idle program to the program to be started. While any control point is running, the exchange package area in its control point area will contain the idle exchange package. An exchange jump to that control point will cause the central processor to re-enter its idle loop and will replace the exchange package in its own control point area. A second exchange jump to initiate the execution of another program will replace its exchange package with the idle exchange package. This procedure will keep the exchange package for each control point in its own control point area except while that program is in execution.

5.7 PPU ASSIGNMENT

If N.PPU is the number of peripheral processors, there are N.PPU-2 pool processors whose assignment is controlled by MTR. (MTR is residing in PPU zero, DSD in PPU number N.PPU-1.)

MTR assigns a PPU by writing a peripheral job name with a control point number in its input register. This is done in the following cases:

a - to satisfy a PP program call issued by a control program (RA+1 request)
b - to answer a PPU request for another PP job (M.RPP or M.RPJ requests)
c - to initiate a stack processor when an I/O request is issued for a mass-storage device to which no stack processor is currently assigned.
d - to call the PP program 1AJ to a control point when all control point activity has ceased.

5.7.1 PPU Status Table

In order to control the PPU assignments, MTR keeps in its memory a PPU status table containing an extended (8 bytes) PPU status word for each PPU.

A - If the PPU is assigned the format of the word is

CPAD	T.PPSx	UNUSED	PPFLAG	PPSEC	PPMSEC

where CPAD is the control point area address of the control point to which the PPU is assigned
T.PPSx is the CMR address of the PPU status word
PPFLAG is a flag telling if the PPU contains a stack processor or not
PPSEC and PPMSEC contain the PP starting time expressed in seconds and milliseconds.

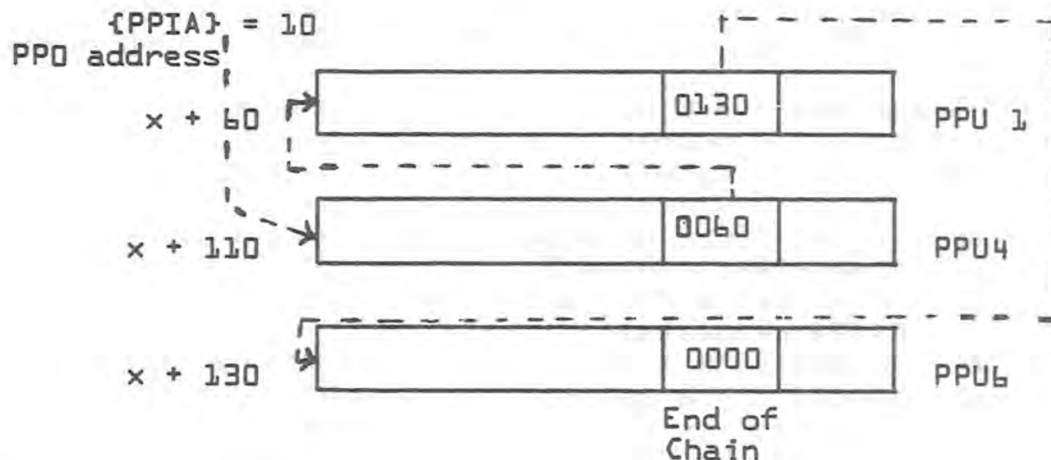
The five first bytes form the CM word written into T.PPSx when PPUx has been assigned.

B - If the PPU is not assigned, it belongs to the chain of unassigned or available PPUs.

The direct cell PPIA points to the beginning of the chain. The fact that the PPU status entry has the same length as the PPU communication area, permits an easy chaining by using the PP input register addresses.

The link is done in byte 5 of the PPU status entry: it contains either the input register address of the next available PPU or zero if no more PPUs are available.

The following diagram showing a chain of three available PPUs explains this linking structure.



Note: There may be one unassigned PPU which is not a member of the available PPU chain. This is the case when no PPU is assigned to a stack processor, MTR reserving one PPU for 1SP.

5.7.2 PP Queue

MTR maintains three different queues:

- a queue of PP jobs which cannot be currently initiated

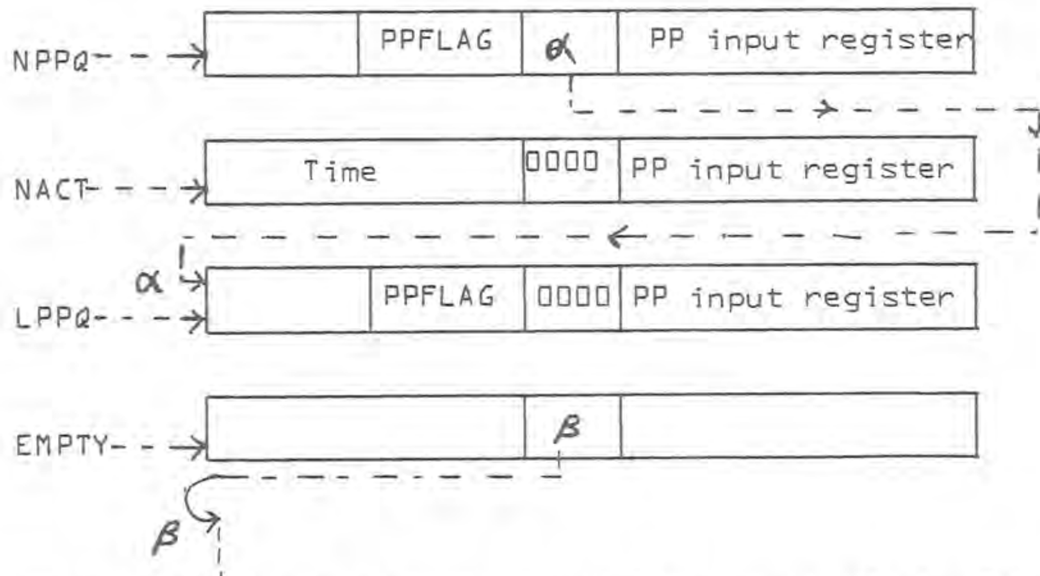
because no PPUs are available. This overlaod queue is in fact a chain of PP input register images.
- a queue of PP jobs which have to be initiated after a certain time delay. This queue, also called delay stack, is time-ordered chain of PP input register images.
- an empty queue which is the chain of all unused entries in the PP queue.

These three chains are all using entries of a single PP queue. There are 24 entries, each entry is 8 bytes long. Three pointers define the beginning of each queue:

NPPQ for the PP job queue
NACT for the PP delay stack
EMPTY for the empty chain

The link is always done by an address in byte 2 (zero means end of chain). The first two bytes are reserved for the mature time in the delay stack, for the PPFLAG (1SP) in the PP job queue. The five remaining bytes contain the PP input register image.

The following diagram shows an example of a PP job queue (2 entries), a PP delay stack (1 entry) and an empty chain.



Note: LPPQ is a pointer to the last entry of the PP job queue. This pointer is necessary because, when the time delay expires, a member of the delay stack will be transferred to the bottom of the PP job queue.

because no PPQ [illegible] queue [illegible] in fact a chain of PP input register images.
- a queue of PP jobs which have to be initiated after a certain time delay. This queue, also [illegible] delay [illegible] is the ordered chain of PP input register images.
- an empty queue which is the chain of [illegible] entries in the PP queue

These three chains [illegible] a [illegible] entries of the [illegible] PP queue. There are 24 entries, each entry is 8 bytes. Three pointers define the location of each queue:

NPPQ for the PP job queue
NACT for the PP delay [illegible]
[illegible] for the [illegible] chain

The [illegible] is always done by an indirect [illegible] byte [illegible] end of chain). The first two bytes are reserved for the mature time in the delay stack, or the PPFLAG [illegible] in the PP job queue. The [illegible] remaining bytes contain the PP input register image.

The following diagram shows an example of a PP job queue (3 entries), a PP delay stack [illegible] and an empty chain.

NPPQ: [illegible] | PPFLAG | [illegible] | PP input register
NACT: Time | [illegible] | PP input register
LPPQ: [illegible] | PPFLAG | 0000 | PP input register
EMPTY: [illegible] | 0 | [illegible]

Note: LPPQ is a pointer to the last entry of the PP job queue. This pointer is necessary because when the time delay expires a member of the delay stack will be transferred to the end of the PP job queue.

Chapter 6--The Displays

		Page
6.0	General	6-1
6.1	Overlay Structure and Organization of Memory	6-3
6.2	Making One's Way Through the Listing	6-5
6.3	General Logic Flow and Main Loop Subroutines	6-5
6.4	Keyboard Input Processing and Command Execution	6-8
6.5	Adding a Command	6-13
6.6	The Displays {General}	6-22
6.7	LOV - The Overlay Loader	6-24
6.8	LDOV - Load Display Overlay	6-24
6.9	The Displays {Individual}	6-25
6.10	Some Notes on the Macros	6-32
6.11	Resident DSD Subroutines and Exits	6-33
6.12	Tables in DSD	6-35
6.13	Locations of Interest	6-35
6.14	The Macros in PPMAC	6-36
6.15	DIS	6-47
6.16	9DM	6-55
6.17	1MH	6-55

6.0 General

The display console is controlled by a display program, DSD, which permanently resides in peripheral processor 9. DSD displays a variety of information concerning the status of the system, including displays of the dayfile, jobs waiting to be executed or printed, etc. DSD permits selected portions of central memory to be displayed and modified if desired. In addition to its display function, DSD processes keyboard entries from the operator. Operator functions include dropping jobs from control points, bringing jobs from the library to a control point, assigning equipment, and selection of displays. DSD also dumps dayfile buffers when requested by MTR.

The main components of the display console are the two cathode ray tubes and the keyboard. By issuing the appropriate function codes to the display console controller, displays of 16, 32, or 64 characters per line may be selected on either the right or left screens. A dot mode display is also available, although only the character mode display is used by the operating system. The display area can be considered to be composed of a grid of points, 512 by 512 points in size. A display can be initiated at any point in the display area by issuing the corrdinates of that point. A vertical, or y, coordinate is sent to the controller in the low-order nine bits of a byte in which the high-order octal digit is a 7. Similarly a horizontal, or x, coordinate is sent to the controller in the low-order nine bits of a byte in which the high-order octal digit is a 6. If the display console controller receives a byte in which the high-order octal digit is neither a 6 or a 7, it is assumed that this byte contains two display code characters.

To display a line of information on the screen, an x and a y coordinate are sent to the controller via the appropriate output instructions (OAN or OAM). These coordinates define the location of the lower left corner of the first character to be displayed. The information to be displayed is then sent to the controller via an OAM instruction. As each character is displayed, the x coordinate is automatically incremented. To display another line, the x and y coordinates should be reinitialized. A coordinate of x = 000 defines the left-most boundary of the display area; a coordinate of y = 776 defines the upper boundary of the display area, and a coordinate of y = 000 defines the lower boundary.

The display should be regenerated between 40 and 50 times per second to avoid flicker. Vertical spacing of the lines should be 5/4 of the horizontal spacing.

Before describing the internal structure of DSD in detail, the hardware considerations and design decisions will be discussed.

Hardware

The 6612 does not have a buffer controller, hence the displays must be repainted 40-50 times per second to maintain suitable intensity.

Space

The major decision was to structure DSD as resident code (0-5700B), and a group of overlays for the displays and commands which are loaded into PP9 for execution. To conserve space in DSD, most of the dayfile processing code was put into MTR, who processes dayfile messages at a faster rate, and the A display was moved out of PP9's memory and back into CM. Putting the A display in PP9's memory had many drawbacks; the most complained about was the shrinking size of the A display as CDC and the users added code to DSD.

In the implementation, the code was written so as to reduce the size of resident DSD and push the code into the overlays whenever possible. Examples are: the structure of the command syntax table (at CMDTBL), the display overlay link table (at DOTBL), the code for the memory display address change commands (Cn,nnnnn.), etc.

Speed

While it is true that DSD wastes a fair amount of time doing nothing when some of the simple displays are up, other displays, such as the memory displays, tend to flicker if the code is not fast enough. For this reason, the octal to display code conversion routines are table driven.

Other examples are the code for the A display and the H display which will flicker under certain extreme conditions. For these displays most of the code is in line and very little checking is done.

Protection of the Scopes

System maladies usually show up on the scopes in one form or another. Extreme system malfunctions may disrupt some of the displays and disable command processing. The following algorithms are implemented:

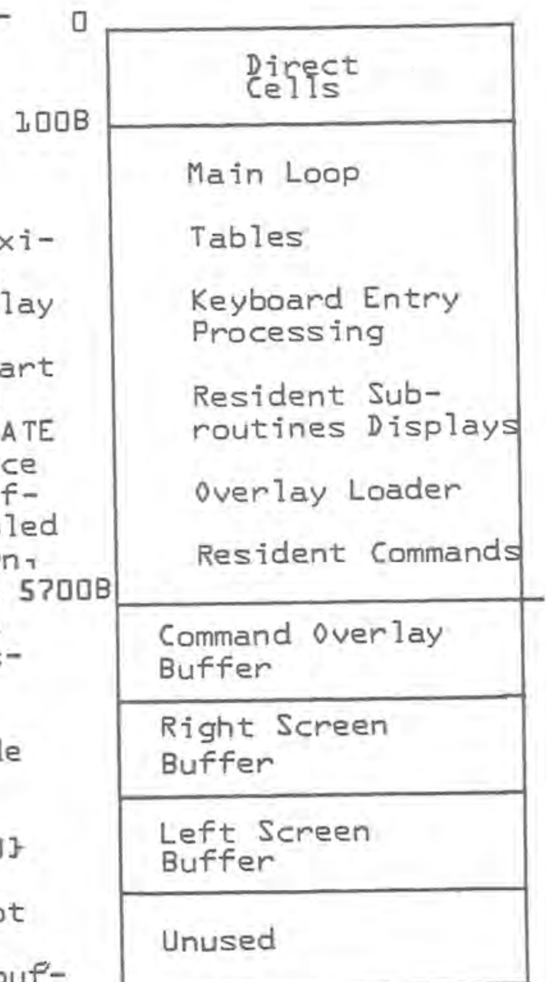
a. Reading up the T. symbols up at deadstart time from CMR and

not referencing them when the system is running. An exception to this rule is T.LIB. If it gets clobbered, one should deadstart.

b. Using counted loops to protect agains a DEAD MTR or clobbered pointers.

6.1 Overlay Structure and Organization of Memory

DSD and all the overlays that are loaded into PP9 are assembled as one program by using the SEGMENT pseudo op. This makes the addresses of the resident subroutines available to the overlays. For simplicity, all of the overlay names begin with 8D. The listing of the display overlays, 8DA - 8DZ follows the listing of the resident code. The overlays 8D1 - 8D9 come last. At present, the overlay 8D9 is unused.



The diagram on the right shows the organization of PP9's memory after deadstart initialization is completed. The deadstart initialization code is overlayed by the command overlay and right screen buffers.

The resident code runs from 100B to approximately 5700B. It is followed by the code for primary initialization, the DATE display and the left screen buffer. The primary initialization is executed once at deadstart and then the space is used to load in the command overlay 8D6 which processes the DATE entry. The DATE display occupies the space that will be used for the right screen buffer. Because the resident code is assembled before the sizes of the overlays are known, one must insert a BSS statement after the primary initialization to ensure that the overlay 8D6 does not overlap the DATE display.

The constraints upon the sizes of the code are:
CMDBUF+CBUFSIZE+Z*DBUFSIZE 7777B
(resident code + buffers fit into the PPU)
LENGTH (8D6) 330B
(Code to process the date command does not overlap the DATE display.)
LWA of DATE display FWA of Left screen buffer.

In addition the size the display buffers should not exceed 601B. The reason for the

constraint lies in the macros used to relocate the M type instructions. One who looks at the REDEF macro will see that the actual constraint is somewhat weaker.

As implied in the above sentences, all the buffer sizes and origins are determined at assembly time and will change when DSD is modified.

Programmers who are making extensive modifications to DSD should monitor the values of the appropriate symbols {listed on the next page} and make adjustments as necessary.

The following symbols are of interest to the programmer:

CMDBUF	End of resident DSD, origin of the command overlay buffer.
CBUFSIZE	Size of the command overlay buffer.
DBUFSIZE	" " " display overlay buffers.
LSBUF	Origin of the left screen display overlay buffer.
RSBUF	Origin of the right screen display overlay buffer.
ENDDSD	LWA+1 of DSD
SPLEFT	Space left for additions and patchs

For the convenience of people modifying DSD these symbols and their values are listed right before the end card.

6.2 Making ones way through the listing

As a convenience to the author and people modifying DSD and DIS, TITLE cards are spread through out the listing. The second page of the listing contains an approximate table of contents which may be used as a guide to people who are searching for any piece of code that falls in some particular subdivision.

6.3 General Logic Flow

DSD is loaded into PP9 at deadstart time starting at location 6. As its first order of business, it jumps to the initialization routine located at the end of the resident code. Here it reads up various constants and pointers from central memory resident and stores them in its memory. After performing the primary initialization, it loads the Z display in the left screen buffer, sets the "DATE" display for the right screen, sets the characters "DATE" in the keyboard buffer and jumps to the main loop. After the operator has successfully entered the date, the B display is brought up on the right screen.

The main loop consists of a series of return jumps to the main subroutines, followed by an "entry point" to the main loop.

We will discuss this section of code first.

Because certain functions which DSD has to perform may take longer than the time allotted for one pass through the master loop, provisions must be made to exit from the subroutine to the main loop (to maintain the displays) and return to the subroutine. Subroutines which must do this are the

overlay loader, and in general, those which interact with the system.

For this purpose, we introduce the "incomplete system transaction flag" (IST), a direct cell, and an entry point in the main loop; MDLOV; to save the return address.

A subroutine which executes a RJM to the main loop (MDLOV) is responsible for first saving any information of interest in a safe place, setting the wait system flag (WS) if a message is to be displayed (e.g. "WAIT SYSTEM DISK"), and then executing the return jump.

The subroutine should also be written in such a way so that no harm will be done if one doesn't return to it. This will happen if the clear or backspace key is depressed. The code is:

```
MDL     ---- BEGINNING OF THE MAIN LOOP

        Main Loop code


        LDD  IST
        ZJN  MDL   loop if no incomplete transaction
        SOD  IST   IST = 0
        LJM  *

MDLOV   EQU  *-1

*       AOD  IST   IST = 1
        UJN  MDL   loop
```

Other things that should be noticed in connection with this are that the keyboard entry processor will accept only the clear or backspace key when IST is set, and that execution of the command will be terminated if they are depressed.

Main loop subroutines.

MDC - Monitor the display channel.

MDC is called once per cycle. Its chief functions consist of ensuring that another PP program (e.g. DIS) may obtain the display channel under appropriate conditions, and calling AVC to adjust the display cycle time for constant intensity. DSD will drop the display channel if the system is not in step mode, the display equipment is assigned and the equipment hold flag is not set.

AVC - Adjust display period.

This subroutine is called from MDC to adjust the display period for constant intensity. This is also necessary to achieve a constant rate for messages that are flashed. The algorithm is read the milli-second clock that MTR updates, compare it to its value last time through the main loop,

delay by calling PDR to process any dayfile requests and loop until 20 milli-seconds have passed. To prevent the display from disappearing in the case that MTR is dead, a count is set and the subroutine exits after 30D times.

DOS - Display One Screen

DOS is called twice from the main loop, once to display each screen. It gets its argument in the A register {screen index and function code}. Functions performed by DOS are to activate the cahnnel, save the function code and the screen index, output the name of the display and control point that it is relative to, call the various subroutines in the screen display table {SDTBL} and disconnect the channel.

On entry to any subroutine in SDTBL, the following conditions hold: the display channel is active and set for small character size, SI holds the screen index {even or ordd}, BA holds the display overlay load address - 77B {for the display overlays}, RCP holds the number of the control point that the display is relative to.

SI and BA are read only outside of DOS, RCP may be reset by a display overlay.

PDR - process dayfile requests.

PDR is called from the main loop and main loop subroutines {DOS, AVC and MDC} to enter stack requests for dayfile buffers that have to be dumped to disk. MTR processes dayfile message requests from the pool PP's. When a message cannot fit in the dayfile buffer, MTR sets bit 8 of byte 1 of the PP's output register and lets the PP hang until DSD clears the bit.

In PDR, DSD scans the output registers of the pool PPU's looking for dayfile functions {m.DFM} that have the bit set. When it finds one it fills out the remainder of the dayfile buffer with words of the form VFD 60/2L+Δ {treated as no space by the printer driver} and attempts to enter a stack request to have the buffer dumped. It is is successful, then it resets the FET and clears the communication bit. Note that one always dumps full PRU's and that MTR will not resume processing of the dayfile request until the FNT status goes inactive.

6.4 PKI - Process Keyboard Input

PKI is called only from the main loop of DSD. Its function is to input characters from the keyboard and process them.

As each character is received from the keyboard, it is checked for a match in the special character and command tables. Special characters, such as backspace, are processed immediately while others are added to the string and the command tables are searched to see if the accumulated entry matches any of the formats in the command table. If there is no match, then the last character inputted is deleted from the string and the message "ILLEGAL ENTRY" is flashed above the keyboard entry line. When enough characters have been entered so that DSD recognizes the accumulated entry as unique, then remaining portions of the entry will be filled in by the keyboard entry processor.

Let us consider an example of the operator assigning a tape.

The operator types in "2." and DSD inputs the characters; notes that there are about 40 commands which start off in this manner (it narrows its search range) and waits for further input. When the operator types in "A", DSD notes that there is now only one command in the table that is of the form n.A and that is "n.ASSIGN uu.", so it fills in the letters "SSIGN". If at this time the operator continues to type in the rest of the word ASSIGN, then the other characters will be rejected because DSD is expecting the EST ordinal (a number) to be entered next. When the operator has entered two octal digits (at least one) DSD notes that the next character is a period, which is "a constant" and it fills it in. Since there are no more characters after this, PKI sets the entry complete flag (KEC) and the keyboard echo line is displayed. Only when the entry is echoed may the operator press the carriage return or repeat mode key. When this is done, the subroutine EXC will be entered to transfer control to appropriate command processor. The command processor will then check the command for legality and execute it if it is legal, otherwise it will jump to one of the error exits.

We will now discuss the subroutines in detail.

PKI

Inputs a character from the keyboard, saves it in PC and checks the incomplete transaction flag. If it is set (we are in the process of loading a disk resident overlay or such) then it jumps to IEXC where we check for a clear or backspace typin, either of which will terminate execution of the command. Otherwise we check the special character tables for a match. Note that one may define special first characters of higher priority then the normal first characters by storing an address into HP.FCT. This is presently done in the N display, and the code in it illustrates the technique. In the case of display overlays, the location will be set to zero when the display is overlaid.

Note that if we find a match in the special character tables, the subroutine SBT immediately jumps to the appropriate processor.

If the character does not match any of those in the special character tables, then we check the central program mode flag {KPM}, which is part of the ='s key table. When in central program mode, we simply add the character to the string since it was not a special character; check for buffer overflow; and exit. In the case of the operator entering too many characters, we delete the character and set the invalid entry flag {KEI}.

Whenever KEI is set, the message that starts at address in KEI will be flashed above the keyboard entry line. The flag is cleared in DOS.

The code starting at PKI3 is the heart of interpretative entry processing. It consists of checking the various flags, searching the command format table, filling in the entry if it is unique, etc.

The best way to form an understanding of the code is to find out how the direct cells are used and how the tables are structured.

The keyboard buffer (KBB) holds the inputted characters, one per byte, right justified. It is always terminated by at least one zero byte. Its length should never exceed 49 unless one makes T.CPOM> 172B and reassembles MTR, DSD and 9DM. The direct cell KS {keyboard start} always holds the address of the keyboard buffer. KI points to first empty byte in buffer when we are not in PKI. In PKI, the pointer will point to the last character entered.

KML, if non-zero, holds the address of a message to be displayed above and on the left of the keyboard entry. It is set when a command processor jumps to one of the error exits. KUM is the unique match flag. It will be set to address of the command format when there is a unique match.

KEC is the keyboard entry complete flag and it will be set non-zero when the keyboard entry is complete. It signifies that the entry is complete and that the operator may now depress the carriage return or repeat mode key. KEC is inspected in the keyboard display subroutine and if non-zero, the keyboard display is echoed. If in the process of executing a command, a command processor jumps to an error exit, KEC will be cleared.

KST is the scan terminated flag and is set; along with KEC; to signify that the entry is complete and that the operator may append any messages that he wishes to the keyboard entry.

K.FM is the first match flag, when non-zero, it holds the address of the first match in the command format table.

Temporaries used by the keyboard processing routines are:

- PC - holds the inputted character
- KFI - holds the number of the character in the command format that we are checking +1 {in CKE and UPC}
- KTA - holds the starting address of the format that we are looking at.
- KMF - is the command match flag and is non-zero when we have found one match in the command format table.

KP - a local pointer to a character in the keyboard buffer (used in CKE, PKI and UPC).

CKE - Check Keyboard Entry

This subroutine is called by PKI to check for a match of a command format to the accumulated keyboard entry. It exits with (A) = 0 if the accumulated keyboard entry matches the command format.

In all cases on exit, KP will point to the first unmatched character in the keyboard buffer and KFI will be set to the first unmatched character in the format string. Both of these quantities must be set if we wish to fill in the keyboard entry. The subroutine consists of a main control loop that compares characters, and 6 sections of code for the special format characters. For example, when we encounter a 71B in the format, we go to COD where we check for an octal field of up to n octal digits where n is specified by the next character in the format.

The subroutine UPC is called by the keyboard processing subroutines to unpack the next character from the format string.

Special Character Processing

Keys such as carriage return, backspace, etc. which require immediate action, are processed specially. They are checked for by calls to SBT from PKI. If a match is found in SBT, then it immediately jumps to the appropriate routine. In most cases, the processing is fairly simple and the comments in the listing should be sufficient.

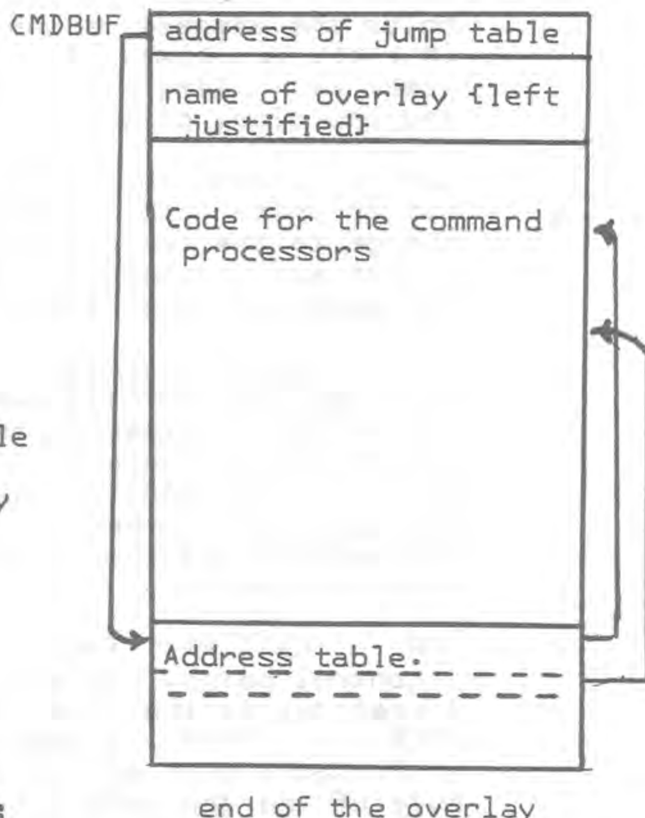
Command Execution

All commands are executed by a return jump to EXC which decides what overlay the command is in, loads it if necessary, and jumps to the appropriate address. For control point commands, the control point number is saved in DCPN and the job name word read into T7 - CM+3, and the first byte is left in the A register for testing by command processors.

Unlike the display overlays, the command overlays are loaded at only one place.

They are structured as follows:

The first byte of the overlay holds the address of the jump table. The second byte of the overlay holds the third letter of the name of the overlay left justified. After this, code to execute the commands appears, followed by the jump table, which holds the address of the entry points for the commands in the overlay.



When the protected mode option is selected, there may be up to 32D commands in an overlay. The commands are identified in the command format table by an ordinal (number of the command in the overlay), and the name of the overlay that they are in.

Command exits are listed on the page before EXC.

Processors that cannot complete their function may save necessary information in a safe place and do a RJM to MDLOV.

Subroutines, etc. for command processing:

SCB - Sets the clear bit at the control point whose number is in DCPN.

The next three subroutines have similar entry and exit conditions.

CDR - Converts a string of octal digits from display code to a binary number.

ACL - Assembles a string of characters left justified. At most, 10 characters will be assembled (packed 2 to a byte).

ASN - Assembles any string of characters between A and 9 left justified. Up to 10 characters will be assembled.

SMF - Send monitor function. This routine is equivalent to R.MTR in pp resident. It will wait approximately 19 milliseconds for the function to be accepted before jumping to an error exit and posting the message MTR DEAD.

DCH - Similar to R.DCH in pp resident, except that no check is made to see if DSD has the channel it is dropping.

CNB - Clear n bytes. A general subroutine to clear n contiguous bytes. This subroutine exists because occasionally a wayward PP will destroy the contents of location zero.* To call this subroutine, one should use the CNB macro (see listing for calling sequence). Note that this subroutine uses T1 and T2 as temporary storage.

*DSD no longer reads up the contents of zero to clear 5 bytes.

RCH - Request channel. Requests the channel whose number is in the A register, for DSD's use.

RPP - Requests that a PPU be assigned. On entry, the A register holds the address of the formatted call. Note that this subroutine, like the mounties, always gets its PPU unless the operator hits the clear key. Safe places to store the formatted call are above 100B and the block of 5 direct cells starting at PPCR.

CCP - Change control point. This subroutine is called when DSD has to write into the storage of a control point. Its function is to change to the control point and stay there if the storage move flag is not set. Otherwise, it switches back to C.P.O.; does an RJM to the mainloop; and repeats the above process.

DFC - Dayfile command. This subroutine is called after a command has been successfully executed to place the command in the dayfile. The method is assemble the command 10 characters per word, left justified and place it in DSD's communication area starting at 164B. It then calls 9DM to pick up the message and sends it to the dayfile. One must use another PP, for if DSD asked MTR to send the message to the dayfile and the message caused a buffer dump....we would have a messy situation.

CPP - Call PP. This is the only subroutine in DSD to call a PPU to a control point. On entry, it expects the name of the program in the A register in the format CAB {ABC is the name of the program}, PPCR + 2 - PPCR + 4 should hold any parameters that are to be passed to PP {bits 0-35 of its input register}. T7 should hold the first byte of the job name {this will be true for control point commands} which should be zero if the PPU program is to be called, and DCPN should hold the control point number. Note that this subroutine will jump to an error exit and post the message ILLEGAL ENTRY if T7 is non-zero. When one wishes to call a PPU to a control point regardless, one should clear T7 first. Examples of calls to CPP can be found in 8D3.

Exits of interest in the resident command processor code are:

DROP1 - Set error flag to F.EROD {operator drop} at the control point whose number is in DCPN and exit.

GO2D - Dayfile a control point command.

6.5 Adding a Command

Before attempting to add a command, one should first read through certain parts of the listing. For convenience these are reproduced in Table 1 the command syntax and address tables, Table 2 the explanation of the syntax language, Table 3 the comments at the end of EXC and Table 4 the comments before command overlay. These tables occur in the listing, approximately on pages 23-25, 35,58,133.

We now concern ourselves with two problems; where to add the code and what to add.

First of all, one should add an entry to the command syntax tables of the form

```
                PROTECT          {optional}

      LNKSYM    CMDTBL  {SYNTAX   comment
```

Kindly maintain the tables in alphabetical order. Please avoid adding control point commands to the table that begin with the letters A or X.

The PROTECT statement should be placed before the command if it is only to be used when the keyboard is unlocked.

LNKSYM is the name of the link symbol that appears on the ▿COMMAND▿ statement. Note that a period is appended to the front of it. SYNTAX is a string of characters which comprise the syntax of the command. Consider the following example.

The format ▿ENPR,≠D4≠,≠FNAME≠, ≥ IOP ≥ .▿ after micro

substitution occurs becomes: ▿ENPR, ↓D, <A >F, ≥IOP≥.▿

It specifies that ▿ the letters ENPR followed by a comma followed by at least one and at most 4 octal digits followed by a comma, followed by a filename {one letter followed by up to 6 letters or numbers}, followed by a comma, followed by one of the letters I,O or P followed by a period▿ is the syntax of the command.

Now comes the problem of where to add the code to execute the command. The choices are as follows:

1. Resident code - If the command is sufficiently important or frequently used.

2. One of the overlays - pick one. Try to pick one so that the overlay 8D9 is presently unused. To create this overlay, one may place the following statements after the ENDC statement in 8D8:

```
    TITLE          8D9  -    USER COMMANDS {or such}

    COMMANDS  9              {command overlay header macro}

LNKSYM  COMMAND.             {command entry point}

    code

    LJM     EXIT.NN          {normal command exit}

    ENDC                     {end of command overlay}
```

3. Part of the code in an overlay and part of the code in 1MH. If the code is very long, then this is the route to go. Examples of code are the EVICT command, the n.X LFNAME and the disk pack commands.

   After deciding the format of the entry, where the code to execute the command should go, etc. we come to the problem of coding the command.

   The code to execute the commands will not be described here because in most cases it is rather short and subject to change. One should note that the philosophy is to push as much code off to other routines as possible.

The macros ˅CMDTBL˅ and ˅COMMAND.˅ are used to make entries in the command syntax tables and to link up the command table entry to the command entry point respectively.

The essential parts {error checking, etc. removed} of the macros are:

```
        MACRO  CMDTBL,LNKSYM,SYNTAX
        VFD  12/.→LNKSYM+PMES.          Form link to command
        DIS  ,$→SYNTAX→$                Form command syntax
 PMES.  SET         0                   Clear protected mode
                                        switch
        ENDM

        MACRO   COMMAND.,LNKSYM

        LOCAL  A
*       form LINK INFO FOR RESIDENT COMMAND TABLE
.→LNKSYM  EQU   CMDINDEX*100B+CMDID  Generate linkage symbol
CMDINDEX   SET   CMDINDEX+1   Advance command index
A      EQU       *             Form ganerated symbol with address
                              of the entry point
CMDTBL   MICRO  1,   ≠ CMDTBL≠A save symbol in string
       ENDM
```

As an example suppose the command n.BLANK appears in overlay 8D3. Then a statement of the form.

N.BLANK COMMAND. will precede the code to execute the command

In the resident command table the statement

```
N.BLANK   CMDTBL ( .BLANK.)
```

will appear.
The above statement will expand to

```
VFD  12/.N.BLANK+PMES.
DIS  $↑.BLANK.$
```

The symbol ▽.N.BLANK▽ will take on the value 0136 which says that it is the second command (01+1) in the overlay 8D3 since 36B = 1R3 in display code.

This process is reversible. Suppose we wished to find out what overlay the code to execute the command ▽n.X LFNAME,FL. is in.

The process is as follows: first we find out the name of the entry in the COMMAND format table. It is ▽N.XEQ▽. We then look up the value of the symbol ▽.N.XEQ▽ in the reference map. Because the first character of all these symbols is a period they will always be the last symbols listed in the reference map. In this case its value is 0042B which means it is the first command in the command overlay 8D7 since 42-33 = 7. One may then skip to the overlay 8D7 using the title cards on the upper left hand corner of the page to find one's way.

In general the value of the symbol is formatted as

```
VFD  1/PB,5/INDEX,6/OVERLAYIN
```

where PB is the protected mode bit. 1 if the command is accessable only when the keyboard is unlocked, else it is 0.

INDEX is the number of the command in the overlay -1, and for resident commands it ranges from 1-37.

OVERLAY IN is 0 for resident commands else it holds the letter of the overlay in display code that the command is in.

TABLE 1

```
                          ************************************************************************ DSD   00693
                          *                                                                        DSD   00694
                          *       THIS MACRO IS TO BE PLACED BEFORE ANY COMMANDS WHICH THE        DSD   00695
                          *       INSTALLATION WISHS TO PROTECT.                                   DSD   00696
                          *                                                                        DSD   00697
                          ************************************************************************ DSD   00698
                                  IFNE    PMEO,0                                                   DSD   00699
                           PROTECT MACRO                  PROTECT A COMMAND                        DSD   00700
                           PMES.   SET    40000                                                    DSD   00701
                                   ENDM                                                            DSD   00702
                                   ELSE                                                            DSD   00703
                           PROTECT OPSYN  NIL                                                      DSD   00704
                                   ENDIF                                                           DSD   00705
                          *                                                                        DSD   00706
               0000000     LLCMD   SET    0               ZERO LENGTH OF LONGEST COMMAND           DSD   00707
               0000000     NCMDS.  SET    0               ZERO NUMBER OF COMMANDS                  DSD   00708
               0000000     PMES.   SET    0               CLEAR PROTECTED MODE ENTRY SWITCH        DSD   00709
                          ***                                                                      DSD   00710
                          *       SOME MICROS TO MAKE KEYPUNCHING EASY                             DSD   00711
                          *                                                                        DSD   00712
                           FNAME   MICRO  1,,*<A>F*       SYNTAX FOR A FILE NAME                   DSD   00713
                           O2      MICRO  1,, +B          SYNTAX FOR TWO OCTAL DIGITS              DSD   00714
                           O4      MICRO  1,, +D                                                   DSD   00715
                           O5      MICRO  1,, +E                                                   DSD   00716
                           O6      MICRO  1,, +F                                                   DSD   00717
                           O20     MICRO  1,, +T                                                   DSD   00718
                           NCP     MICRO  NCP,1,*1234567*  NUMBER OF CONTROL POINTS                DSD   00719
                           LSN     MICRO  LSDN-1,1,*ABCDEFGHIJKLMNOPQRSTUVWXYZ*                    DSD   00720


                          ***                                                                      DSD   00722
                          *       THIS TABLE SPECIFIES THE SYNTAX OF THE COMMANDS THAT CAN BE      DSD   00723
                          *       ENTERED FROM THE KEYBOARD.  THE SPECIAL CHARACTERS FROM + TO ≡   DSD   00724
                          *       ARE USED TO SPECIFY SYNTAX CLASSES, ETC.  A FULL TABLE OF THE    DSD   00725
                          *       CHARACTERS USED AND THEIR MEANING CAN BE FOUND AT THE BEGINNING  DSD   00726
                          *       OF THE CODE FOR KEYBOARD ENTRY PROCESSING.                       DSD   00727
                          *$$$    PLEASE KEEP TABLE IN ALPHABETICAL ORDER ( FOR THE Y DISPLAY )    DSD   00728
                          *                                                                        DSD   00729
               0000360     CMDTBL  EQU    *               COMMAND TABLE START                      DSD   00730
                          **                                                                       DSD   00731
                          **      SYSTEM COMMANDS                                                  DSD   00732
                          **                                                                       DSD   00733
 0350                      DSC     CMDTBL (<9.)           XY. OR X. ( SET LEFT AND/OR RIGHT        DSD   00734
                          *                               SCREEN DISPLAYS. )                       DSD   00735
 0364                      SDCPN   CMDTBL (≤A≠LSN≠=+A.)   SET DISPLAY TO C.P. N                    DSD   00736
 0372                      ACK     CMDTBL (ACK.)          ACKNOWLEDGE A C.P. ZERO MESSAGE          DSD   00737
 0376                      ACNXX   CMDTBL (ACN≠O2≠.)                                               DSD   00738
 0403                      AUTO    CMDTBL (AUTO.)         AUTO.                                    DSD   00739
 0410                              PROTECT                                                         DSD   00740
 0410                      CALL    CMDTBL (CALL,>C,<0≠NCP≠,+L.)  CALL,NAM,N,NNNN NNNN NNNN.        DSD   00741
                                   IFNE   S20,0,2                                                  DSD   00742
                                   PROTECT                                                         DSD   00743
                           CTRAP   CMDTBL (CTRAP,>C≤07)   CTRAP,NAM N                              DSD   00744
               0000422     DATEMT  EQU    *                                                        DSD   00745
 0422                      DATE    CMDTBL (DATE ≠DATMIC≠.)   DATE MM/DD/YY.                        DSD   00746
 0441                      DCNXX   CMDTBL (DCN≠O2≠.)                                               DSD   00747
```

TABLE 1-


```
0446                ENPR     CMDTBL (ENPR,≠04≠,≠FNAME≠,≥IOP≥.)      ENTER FILE PRIORITY       DSD      00748
0452                EVICT    CMDTBL (EVICT,≠FNAME≠,≥IOP≥.)     EVICT,FNAME,T.                 DSD      00749
0475                FANXX    CMDTBL (FAN≠02≠.)                                                DSD      00750
0502                FCNXX    CMDTBL (FCN≠02≠≥.,≥≠04≠.)    FCN XX. OR FCN XX,YYYY.             DSD      00751
0512                IANXX    CMDTBL (IAN≠02≠.)                                                DSD      00752
0517                LDAXY    CMDTBL (LDA≠04≠.)         LDA NNNN.                              DSD      00753
          0000525   XYLEFT   EQU    *+1                                                       DSD      00754
0524                LEFT=    CMDTBL (LEFT=<H.)         LEFT=LLLLLLLL.                         DSD      00755
                             IFNE   PMEO,0,2                                                  DSD      00756
0532                         PROTECT                                                          DSD      00757
0532                LOCK     CMDTBL (LOCK.)            LOCK. THE KEYBOARD                     DSD      00758
0537                MCHXX    CMDTBL (MCH≠02≠.)                                                DSD      00759
0544                OANXX    CMDTBL (OAN≠02≠.)                                                DSD      00760
                             IFNE   MCPU,1,1                                                  DSD      00761
                    OFFCPU   CMDTBL (OFFCPU<AB.)       OFF CPU X.                             DSD      00762
0551                OFFXX    CMDTBL (OFF≠02≠.)         OFF NN.                                DSD      00763
                             IFNE   MCPU,1,1                                                  DSD      00764
                    ONCPU    CMDTBL (ONCPU<AB.)        ON CPU X.                              DSD      00765
0556                ONXX     CMDTBL (ON≠02≠.)          ON NN.                                 DSD      00766
                             IFNE   S1D,0,2                                                   DSD      00767
0563                         PROTECT                                                          DSD      00768
0563                PPN      CMDTBL (PP≤08)            PP N                                   DSD      00769
0570                SEQ      CMDTBL (SEQ,<D≥,.>≠04≠≥,.≥≠02≠.)   SEQUENCER COMMANDS            DSD      00770
0604                         PROTECT                                                          DSD      00771
0604                STEP     CMDTBL (STEP.)            STEP.                                  DSD      00772
                             IFNE   S2D,0,2                                                   DSD      00773
                             PROTECT                                                          DSD      00774
                    SETRAP   CMDTBL (STRAP,>C≤07)      STRAP,NAM N                            DSD      00775
0611                TIME     CMDTBL (TIME.≤02≤09.≤05≤09.≤05≤09.)    ENTER TIME.HH.MM.SS      DSD      00776
                             IFNE   PMEO,0,2                                                  DSD      00777
          0000630   AUNLF    EQU    *                                                         DSD      00778
0630                UNLOCK   CMDTBL (UNLOCK,<C.)       UNLOCK,≠IPW≠.                          DSD      00779
                    *                                                                         DSD      00780
0637                CEDP     CMDTBL (E,<B.)            E,XX. OR E,X.                          DSD      00781
0644                CHDS     CMDTBL (H,≥IOCLP≥≥IOCLP.≥.)  CHANGE THE H DISPLAY                DSD      00782
0657                CMDC     CMDTBL (>CDG≥≤04,+F.)     CN,NNNNNN.                             DSD      00783
0657                         PROTECT                                                          DSD      00784
0667                ENM      CMDTBL (≠06≠,≠020≠.)      ENTER MEMORY                           DSD      00785
0674                         PROTECT                                                          DSD      00786
0674                ENM      CMDTBL (≠06≠,≤04,≠04≠.)   MODIFY A BYTE                          DSD      00787
0703                         PROTECT                                                          DSD      00788
0703                ASCST    CMDTBL (≠06≠/")           MIXED FORMAT MEMORY ENTRY              DSD      00789
                    **                                                                        DSD      00790
                    **      JANUS COMMANDS                                                    DSD      00791
                    **                                                                        DSD      00792
0707                JANCMD   CMDTBL (/ABORT.)          DROP EVERYTHING IMMEDIATELY            SCP316L  00001
0715                JANCMD   CMDTBL (/BS≠02≠>.,≥≠02≠≥.,≥≤A=)  BS UU,PP,C                      DSD      00793
0730                JANCMD   CMDTBL (/END≠02≠.)        /END UU.                               DSD      00794
0736                JANCMD   CMDTBL (/OK≠02≠.)         /OK UU.                                DSD      00795
0743                JANCMD   CMDTBL (/REP≠02≠.)        /REP UU.                               DSD      00796
0751                JANCMD   CMDTBL (/REW≠02≠.)        /REW UU.                               DSD      00797
0757                JANCMD   CMDTBL (/SUP≠02≠.)        /SUP UU.                               DSD      00798
0765                JANCMD   CMDTBL (/SW≠02≠.)         /SW UU.                                DSD      00799
```

TABLE1-b


```
                0000772    CPCTBL   EQU    *                        START OF THE CONTROL POINT COMMANDS   DSD      00801
                           **                                                                             DSD      00802
                           **       CONTROL POINT COMMANDS                                                DSD      00803
                           **                                                                             DSD      00804
0772                       N.ASS    CMDTBL (*.ASSIGN ≠02≠.)  N.ASSIGN UJ.                                 SCP316L  00002
                                    IFEQ   IP.DPAK,0,2                                                    DSD      00806
1002                       N.BLANK  CMDTBL (*.BLANK    ≠)   N.BLANK.                                      DSD      00807
                                    IFCP   1                                                              DSD      00808
                           N.BLANK  CMDTBL (*.BLANK≥.,≥≠02≠.)  N.BLANK. OR N.BLANK,UU.                    DSD      00809
1011                       N.CFO    CMDTBL (*.CFO %)         N.CFO (MESSAGE)                              DSD      00810
1017                       N.CKPT   CMDTBL (*.CHECKPT.)      N.CHECKPT.                                   DSD      00811
1026                       N.CLEAR  CMDTBL (*.CLEAR.)        N.CLEAR.                                     DSD      00812
1034                       N.COM    CMDTBL (*.COMMENT %)     N.COMMENT (MESSAGE)                          DSD      00813
1044                       N.DFLE   CMDTBL (*.DAYFILE,<3.)   N.DAYFILE,LL.  LL = LP / CP / MT             DSD      00814
                                    IFNE   DPAK,0,1                                                       DSD      00815
                           N.PDPC   CMDTBL (*.DEVADD≠02≠.)   N.DEVADD UU.                                 DSD      00816
1055                       N.DEVT   CMDTBL (*.DEVTYPE A≥ABCDPX*≥≥.,≥≠02≠.)   N.DEVTYPE AL,02.             DSD      00817
1075                       N.DIS    CMDTBL (*.DIS.)                                                       DSD      00818
1102                       N.DUF    CMDTBL (*.DMPQ,≥IOP≥.)   N.DMPQ,T.                                    DSD      00819
1113                       N.DROP   CMDTBL (*.DROP.)                                                      DSD      00820
1121                       N.DUMP   CMDTBL (*.DUMP.)         N.DUMP.                                      DSD      00821
1127                       N.PRTL   CMDTBL (*.ENPR,≠04≠.)    N.ENPR,NNNN.                                 DSD      00822
1136                       N.PRTL   CMDTBL (*.ENTL,≠05≠.)    N.ENTL,NNNNN.                                DSD      00823
1145                       N.GO     CMDTBL (*.GO.)           N.GO.                                        SCP316A  00367
1152                       N.JANUS  CMDTBL (*.JANUS.)        N.JANUS.                                     DSD      00825
1150                       N.KILL   CMDTBL (*.KILL.)                                                      DSD      00826
1166                       N.LOAD   CMDTBL (*.LOAD≥.X≥.)     N.LOAD. AND N.LOADX                          DSD      00827
1176                       N.NEXT   CMDTBL (*.NEXT.)         N.NEXT.                                      DSD      00828
1204                       N.ONFSW  CMDTBL (*.OFFSW≤16.)     N.OFFSW N.                                   DSD      00829
1214                       N.ONFSW  CMDTBL (*.ONSW ≤16.)     N.ONSW N.                                    DSD      00830
1224                       N.RECK   CMDTBL (*.RECHECK.)      N.RECHECK.                                   DSD      00831
                                    IFNE   MCPU,1,1                                                       DSD      00832
                           N.RELCP  CMDTBL (*.RELCPU.)       N.RELCPU                                     DSD      00833
1233                       N.RERUN  CMDTBL (*.RERUN≥.,≥≠04≠.)  N.RERUN. OR N.RERUN,PPPP.                  DSD      00834
1244                       N.QUE    CMDTBL (*.RESQ,≥IOP≥.)   N.RESQ,T.                                    DSD      00835
1255                       N.ROLL   CMDTBL (*.ROLL≥IO≥.)     N.ROLLT. AND N.ROLLO.                        DSD      00836
1265                                PROTECT                                                               DSD      00837
1265                       N.STEP   CMDTBL (*.STEP.)         N.STEP.                                      DSD      00838
                                    IFNE   DPAK,0,1                                                       DSD      00839
                           N.PDPC   CMDTBL (*.UNLOAD≠02≠.)   N.UNLOAD UU.                                 DSD      00840
                                    IFNE   MCPU,1,1                                                       DSD      00841
                           N.USECP  CMDTBL (*.USECPU≤A9.)    N.USE CPU X.                                 DSD      00842
1273                       N.VRN    CMDTBL (*.VRN,>F.)       N.VRN,XXXXXX.                                DSD      00843
                           *                                                                              DSD      00844
1302                       N.XFQ    CMDTBL (*.X ≠FNAME≠≥.,≥≠06≠.)   N.X FNAME,FL.                         DSD      00845
1314                                PROTECT                                                               DSD      00846
1314                       N.MEM    CMDTBL (*.≠06≠,≠020≠.)   ENTER MEMORY RELATIVE TO A C.P.              DSD      00847
1322                                PROTECT                                                               DSD      00848
1322                       N.MEM    CMDTBL (*.≠06≠,≤34,≠04≠.)  MODIFY A BYTE                              DSD      00849
                           *                                                                              DSD      00850
1332       0000                     DATA   0                 END OF THE TABLE                             DSD      00851
1333       0000            CMDLNK   DATA   0                 ADDRESS LINK TO NEXT TABLE                   DSD      00852
```

TABLE II

KEYBOARD ENTRY PROCESSING WORKS IN AN INTPRETATIVE MANNER. AS EACH CHARACTER IS ENTERED FROM THE CONSOLE IT IS CHECKED AGAINST THE ENTIRE LIST OF POSSIBLE COMMANDS. IF NO MATCH IS FOUND, THEN THE CHARACTER IS REJECTED AND THE MESSAGE ≠ILLEGAL ENTRY≠ IS FLASHED ON THE CONSOLE. IF A MATCH IS FOUND, THEN THE CHARACTER IS ADDED TO THE STRING AND THE SEARCH RANGE NARROWED. WHEN ENOUGH CHARACTERS HAVE BEEN ENTERED SO THAT THE COMMAND IS UNIQUE THE KEYBOARD ENTRY ROUTINE FILLS IN THE REST OF THE STRING UP TO A SPECIAL FORMAT CHARACTER.

TABLE OF SPECIAL FORMAT CHARACTERS AND THEIR MEANINGS

70B ↑ { 11-8-5 } CONTROL POINT NUMBER 1≤C≤N.CP

71B ↓ { 11-8-6 } AN OCTAL FIELD 0≤C≤7
ALWAYS FOLLOWED BY A FIELD COUNT
NOTE THAT EMBEDDED BLANKS ARE ALLOWED

72B < { 12-0 } ALPHABETIC FIELD A≤C≤Z
ALWAYS FOLLOWED BY A FIELD COUNT

73B > { 11-8-7} ALPHA/NUMERIC FIELD A≤C≤9
ALWAYS FOLLOWED BY A FIELD COUNT

74B ≤ { 8-5 } INTERVAL DELIMETER ≤ XY MEANS C IS LEGAL
IF X≤C≤Y

75B ≥ {-12-8-5 } SET DELIMETER ≥XYZ≥ MEANS C IS LEGAL
IF C BELONGS TO THE SET

IF A PERIOD { . } OCCURS IN THE SET AND THEIR IS A MATCH WITH IT, THEN THE ENTRY IS CONSIDERED COMPLETE AND THE REST OF THE FORMAT IS IGNORED.

76B { 12-8-6 } TERMINATE SCAN - THE ENTRY IS UNIQUE AND
ANY STRING MAY FOLLOW.
USED FOR COMMANDS THAT MAY BE COMMENTED.

TABLE III

** ENTRY CONDITIONS FOR COMMAND EXECUTION **

```
A)  ALL COMMANDS     - KEYBOARD ENTRY IN THE KEYBOARD BUFFER,
                       KBB, AND KI POINTS PAST THE LAST
                       CHARACTER OF THE COMMAND.
B)  SYSTEM COMMANDS - NO ADDITIONAL INFORMATION
C)  C.P. COMMANDS    - CONTROL POINT NUMBER IN DCPN, T7 - CM+3
                       HOLDS THE JOB NAME WORD AND (A) = (T7)
                       ON ENTRY TO THE COMMAND.
```

(T7) SHOULD BE PRESERVED IF YOU WISH TO CALL A PP PROGRAM TO AN EMPTY C.P. . FOR FURTHER INFO SEE THE CPP SUBROUTINE AND CODE IN 8D3.

TABLE IV

SOME NOTES ON THE COMMAND OVERLAYS

A) FOR A DESCRIPTION OF GENERAL ENTRY CONDITIONS SEE THE COMMENTS AFTER SUBROUTINE ≠EXC≠.

B) BECAUSE SPACE IN THE COMMAND OVERLAYS IS LIMITED ONE SHOULD CODE CAREFULLY. THE DIRECT CELLS ARE USED AS FOLLOWS:
1 - 21B ARE GENERAL TEMPORARIES
IN PARTICULAR, MTR REQUESTS GO IN CM - CM+4,
AB - AB+4 IS USED AS AN ASSEMBLY BUFFER BY CDR, ASN AND ACL.
THE IMAGE OF A PP≠S INPUT REGISTER GOES IN PPCR - PPCR+4.
THESE 5 DIRECT CELLS WILL BE PRESERVED IF A SUBROUTINE THAT THE CODE CALLS DOES A RJM TO THE MAIN LOOP.

C) IF THE CODE YOU ARE WRITING IS VERY LARGE, REMEMBER THAT THERE IS PLENTY OF SPACE IN 1MH.

D) REMEMBER TO SAVE QUANTITIES OF INTEREST, ETC. IN A SAFE PLACE BEFORE CALLING A SUBROUTINE THAT DOES AN RJM TO THE MAIN LOOP.

E) REMEMBER THAT OVERLAYS IN DSD ARE NOT RELOADED. KEEP THIS IN MIND WHEN YOU WRITE CODE THAT MODIFIES ITSELF.

F) ABOVE ALL, KEEP IN MIND THAT THE KEYBOARD ENTRY ROUTINE HAS PERFORMED A COMPLETE SYNTAX CHECK ON THE COMMAND AND IN MOST CASES IT IS NOT NECESSARY TO CHECK FOR TERMINATING PERIODS, ETC.

6.6 The Displays

Before attempting to add a display one should read the documentation on DOS, the description of display overlay structure and the macros used {Pages 74-78}, the listing of the basic display subroutines and macros {Pages 30-34}, and the macros, most of which are used in the display overlays {Pages 10-12}.

First of all one should change LSDN at DSD.247, which is the number of the last sequential display +1, so that entries will be made in DOTBL to link the display up, the size of the ='s key table increased, etc. Next one should add a statement to the Z display table ZDMS that describes the display.

Assume we are adding the P display. The basic skelton of the display consists of the following:

```
          TITLE     P  DISPLAY - THE FRAMZEL TABLE

          DISOVL    P  {specifies start of the overlay named 8DP

          DISHDT    P, arguments {provides link info for the display
                                  loader}

          SPACE 3
*
*         P display initialization {optional}
*
   INTP   .
          .
          .
          code
          .
          LJM   LDOVXX  exit to the display loader

*
*
*         + - key processing     {optional}
*
*
*
*         advance table address by 4 gunillas
*         .
   PMKPP  .     code is entered here
          .
          .
          code
          .
          LJM   RMK2    exit to clear out the key
*
*
          EENM  DSPP    display entry/exit line
          .
          code to execute the display
          .
          Return

          Subroutines

ODBUFFER  OVBUF     10D    10 byte buffer for the use of the P display
*
          DEND  {end of display overlay card forms relocation tables, etc.}
```

On entry to a display overlay, the display channel has been functioned for small character size, RCP holds the number of the control point that the display is relative to, etc. The direct cells are used as follows:

0 is not used

1-21B (T1-AB+4) are used as general temporaries. They may be used freely to hold pointers, loop counts etc. Most of the subroutines that are called by the display overlays (D2, D4, D5, CNB, Z6, etc.) use T1-T3 as temporary storage, so one should be careful in using them.

RCP, as mentioned before holds the number of the control point that the display is relative to. It may be stored into if the ='s key is not defined for the display.

The cycle counter, CC, is a 12 bit counter that is incremented by one each time around the main loop. It is inspected by subroutines that flash and pulse messages.

XC, holds the constant 6000B, an X coordinate of 0, and is read only.

YC is usually used to hold a Y coordinate, but may be used for other purposes.

RF, the repeat flag is used as index for loop counts (count down) by the macros RDLP and RDLT.

Other direct cells which may be of interest to the displays are:

NCPS - holds the number of control points + 1

DFBA - holds (FWA of the dayfile buffer area) /10B

FNTA - holds FWA+4 of the file name table

LFNT - holds the LWA+1 of the FNT

TEST - holds the FWA of the EST

LEST - holds the LWA+1 of the EST

The following should be kept in mind when writing a display overlay.

The size of the overlay should not exceed 601B (change the redef macro if it does). The display must be repainted approximately 50 times a second, hence the time it takes to generate a display should not exceed 10-15 milli-seconds. This is ample time in most cases. A quick test of the speed of the display can be made by bringing it up on both screens.

When using medium character size, the characters should be regenerated twice to achieve the same intensity as small character size.

6.7 LOV - The overlay loader

LOV is called by EXC and LDOV to load command and display overlays. In essence it is similar to R.OVL in pp resident. Because the time it takes to load a disk resident overlay is longer than display cycle time, the subroutine is phased.

The first phase consists of saving the load address, checking for overlay in DSD; we exit with A = 0 if it is already in, and searching the library for the overlay. In the library search portion we inspect the first 256 entries for the overlay we are looking for. Putting a maximum limit on the search prevents DSD from hanging up in case the library pointer has been clobbered. If the overlay is CM resident, then we read it up and exit. For overlays that reside on a mass storage device we enter phase 2 of the overlay loading process. This consists of calling 9DM to control point zero to load the overlay for us, and waiting until it is ready to transmit. 9DM will signal that it is ready to transmit the overlay by clearing the second byte of its input register. After it does this, it will wait approximately 120 milliseconds (6 cycles around DSD's loop) for a response. If there is no response from DSD in that time (DSD sets byte 2 to a 1) then it drops the ppu.

Phase three of the overlay loader consists of setting up the display channel for 9DM to transmit the overlay across it, waiting for 9DM to start transmission and finally, inputting the overlay.

One should note that in reading the overlay in from CM or inputting the overlay across channel the first 5 bytes of the overlay are dropped. This is because they contain header information which is duplicated in the second byte of the overlay.

Also note that the overlays are not checksummed after they are read in. The reasons for not putting this feature in were: in the case of disk resident overlays it is unnecessary; in the case of CM resident overlays it doesn't really do any good since one will have to deadstart if the message "8DX - BAD CHKSUM" appeared on the console; and finally it would take approximately 100B bytes of space to do the checksum properly.

6.8 LDOV - The display overlay loader

LDOV is called to load and link up display overlays to DSD. The parts of the routines are:

a. look up and save the name of the overlay that the display is in along with the ordinal to the link table.

b. delink any table extensions, clear address of +- key routine, etc.

c. call LDOV to load the overlay

d. If the overlay has just been loaded into DSD's memory, then relocate it.

e. Link the display to DSD. This consists of storing the addresses of the entry point in the SDTBL, the address of

the +- key routine in PMKA, the name of the display in LST or RST.

f. Finally, if the display is to be initialized, then we jump to the initialization routine and exit.

The two tables involved in loading a display overlay are DOTBL, which tells us the ordinal to the link table and the name of the overlay that the display is in, and the link table. Suppose we wish to bring the J display. At DOTBL + 1RJ we find that it is in the overlay 8DA and the ordinal to the link table is 12B. After loading the overlay we extract the information from the link table which starts at the load address + 12B. The information there is

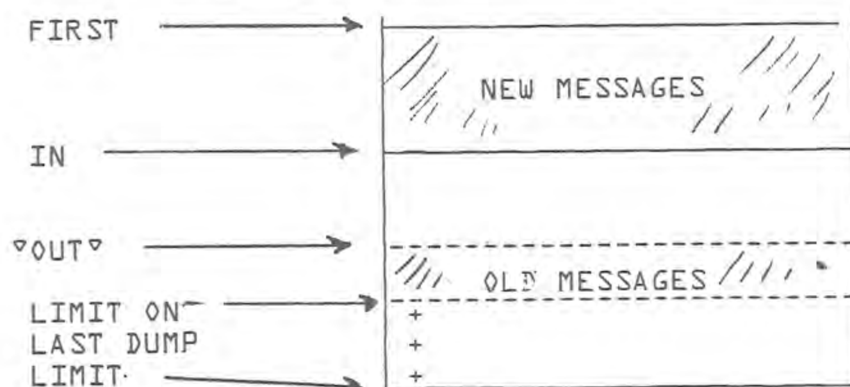
```
    vfd 12/name of display {only if ='s key defined}
    vfd 12/ordinal to +- key routine or 0 if none
    vfd 12/ordinal to address of entry point
    vfd 12/ordinal to address of initialization routine
        or zero
```

6.9 The A Display

The A display is a display of the system or control point dayfile buffers. First we will discuss the structure of buffers and FETs. The dayfile buffers are handled in a linear fashion by MTR who inserts messages at IN if there is room. When there is no room or a buffer dump is called for, MTR calls DSD to dump the buffer. DSD, before entering a stack request to dump the buffer, fills the unused portion of the buffer from IN to LIMIT-1 with words of the form 60/2L+ ∆ , which are treated as a "no space" code by the printers.

The FETs for the dayfile buffers are one word affairs and are structured as follows: 12/IN-FIRST, 12/LIMIT-FIRST, 12/FIRST-T.DFB, 12/LIMIT ON LAST DUMP - FIRST, 12/0.

The reason for not keeping relative quantities and not the absolute quantities in the FET is that T.DFB may be greater than 10,000B. The limit on the last dump is simply the value of IN at the time the buffer is dumped.



To display the contents of the buffer in such a fashion that the older messages are displayed at the top of the screen and the newer messages at the bottom, one starts at IN, skips past a partial message and calls that "out". One then displays the contents of the buffer from OUT to LIMIT (until we hit a word of the form 60/2L+Δ) and from FIRST to IN or the bottom of the screen. That is, if we hit the bottom of the screen before we hit OUT, then we "roll the display". The rolling effect is achieved by advancing OUT towards LIMIT by skipping past a message and moving the Y coordinate down a line. During the next 10 times through the display the Y coordinate is advanced by 1 until we reach the top. The rolling process continues until we hit IN before we reach the bottom of the screen.

The "bottom of the screen" is determined during deadstart initialization from the size of the buffer. The formula is Top Y coordinate - Bottom = Buffer size * 10/7, where 7 is the maximum number of words in a message.

One should note that since MTR can add messages to the display at a rate faster than DSD can roll the display, the display may temporarily "disappear" under heavy dayfile message processing loads.

The B Display

This display shows the statuses of all the control points. The essence of the display is to read up information from the control point areas, convert it and display it. For each control point 6 lines of information are displayed, except for JANUS control points, which may have up to 7 lines of information displayed. The specs for the display are given in the listing and here we restrict ourselves to a discussion of the fourth and fifth lines of the display which are the B display message lines. Words C.CPDFM - W.CPDFM+7 (30B - 37B) of the control point area. The first line is 5 words long and the second 3. Dayfile messages that go to the B display will be sent to this area, starting at W.CPDFM.

The second message line is usually used by the PPU's to send trouble messages to the operator, such as: "MT XX REJECT", "WAIT - DISK FULL", etc.

Messages going to either of the lines will be flashed, pulsed or intensified if the pause bit at the control point is set or the first character of the message is a $.

The Central Memory Displays

The central memory display are the simplest of the displays. They consist of 4 matrices of 8 lines, where each line consists of the relative address of the word, the contents of the word (in octal) and the BCD equivalent.

The main subroutine that drives all the displays is DCM. Each display consists of giving the address of the octal conversion word format subroutine and the address of the table that gives the words to be displayed, followed by RJM DCM.

The format of the address table is vfd 24/address, 24/address, 24/address, 24/address.

The E and F displays are rather simple and will not be discussed here.

The H Display

The H display shows the files according to their type with some of their basic attributes of interest listed. For input files the time limit, field length, and priority is shown. For output files, the file size (in sectors) and the job priority is shown.

At the top of the screen the available FNT count and the amount of space allotted for RBT storage is shown.

The code consists of setting up the direct cells, outputting the headers, looping through the FNT for files of the appropriate type and displaying the information until both columns are full.

The K Display

For a non-empty control point, with non-zero priority the K display is similar to B display for single control point except that the processor status is not displayed and the next 20 upcoming control cards are displayed. For all other cases the FWA and LWA+1 of some of the system tables will be displayed. This display is table driven and the macro CMRPD is used to generate the table.

The format of the table is 4 bytes per entry where each entry is of the form

```
vfd 24/Name in display code
vfd 6/byte address of the pointer after executing a CRD CM
    instruction
vfd 6/P.Name address of word that holds the pointer
vfd 6/1+LWA flag (LWA flag = 0 if no LWA is to be displayed)
vfd 6/shift count for positioning pointers that are larger than
    12 bits.
```

The arguments to the macro are

Name	-	the name of the table	(EST)
SIZE	-	the size of the pointer in bits	(12)
LWAF	-	if the LWA+1 of the table is to be displayed	(1)

PDSTFF - Non blank if there is no C. symbol in SCPTEXT {0}
for the pointer. The value should be that of
the C. symbol, if it existed.

The L Display - Central Programmable

The L display is the left tube central programmable display. When it is on the left screen it may be attached to a control point that is formatted properly so that information from a buffer in the control points field length may be displayed and the operator may communicate with a CPU program at the control point.

The specifications are as follows:

For the control point to be considered as an L display control point RA+70B must contain 8RLDISPLAY. If this is so then DSD will assume that the information to be displayed begins at RA+110B, that RA+100B-RA+106B, is a communication buffer for messages from the console and when the control point is not being moved byte 4 of RA+107B will hold a 12 bit cycle count that is updated by DSD once each time through its master loop {approximately 50_{10} times per second}.

The reasons for placing the buffers in fixed places is for simplicity and speed on DSD;s part. Programmers should note that labelled common gets loaded starting at RA+100B and the statement COMMON/DSDCOMM/KEYBC{7},ICYCLE,DISBUF{484 will create the appropriate buffers.

For the purposes of display the information in the buffer is broken up into lines. Each line consists of information such as length of the line in words, character size to be used number of times the line is to be displayed per cycle, coordinates and characters.

The format of a line is:

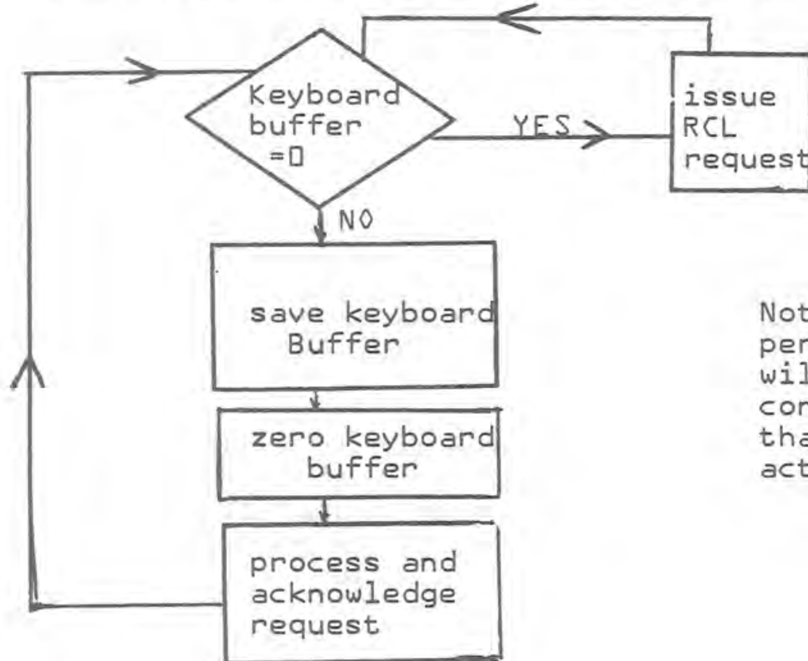
6/word count {≤16, 0 or >16 acts as a buffer terminator}
3/character size {0, 1 or 2 for small, medium or large}
3/repeat count {if 0, then 1 is assumed}

The rest of the line is interpreted as coordinates and BCD data, which may appear in any order. Note that the central program is responsible for supplying coordinates. One may break the display up into a grid that consists of 51 lines and 64 columns. The spacing between columns is 8 and lines 10. The area that the central program may use lines above line 4 and below line 48.

The following COMPASS code illustrates two sample lines.

```
+   vfd  6/ENDL1-*,6/0,12/6000B  info byte and x coordinate
    vfd  12/7000B+47*10   Y coordinate, line 47
    DIS  ,* This should appear on line 47*
ENDL1 EQU *
+   vfd  6/ENDL2-*6/0,12/6000B
    12/7000B+20      Line 2
    ,*FORMAT ERROR*
ENDL2  EQU *
```

A CPU program which wishes to communicate with DSD should have a main loop which may be flow charted as follows:



Note that one must go into periodic recall since MTR will attempt to abort a control point in auto recall that doesn't have any activity at it.

When the L display is at appropriate control point the message ▽PROGRAM MODE▽ will appear above the keyboard entry buffer.

In program mode DSD doesn't do any interpretive keyboard processing. The characters are collected and when the carriage return key is depressed the accumulated entry is packed 10 characters per word, left justified and transferred to the central program starting at RA+100B. Other keys which will be interpretated in central program mode are the + and —, backspace, space, ='s and * key.

The M Display

This is a display of the first four words of each PPU's communication area. To the right of the display of the PPU's output register is the last monitor function that the PPU issued. In the case that the PPU issued a bad MTR function the message ▽PP HUNG▽ is flashed. The M display, like most other displays

is relatively simple and will not be discussed any further here.

The N Display

The N display formats and displays the DSD/1DP communication area. 1DP is a separate PPU program, along with some mods to CMR and STL which permit the debugging of PPU programs on line. After a program is brought under the control of the debugging processor (1DP) its activity may be monitored from the N display. A PPU program under debug control may be stepped one instruction at a time, run free with central write stops, breakpoints, etc. set. Note that 1DP is not a standard product and it and associated mods and documentation may be obtained through VIM.

The O Display

The display part of the O display is similar to the Z display. The bulk of the O display consists of checking the keyboard entry processing flags to see if the operator is typing in a command of the format n*nn, and if he is, filling in the appropriate message from the operator message table in the display.

One may wonder why the messages were put in a display instead of the command table. The answers are that the command table was getting rather large and that some installations might wish to disregard the display and would be annoyed by the space that it took up in the resident code.

The Y Display

The Y display is a decoding of the command format tables into a form that is displayable and hopefully understandable. The code consists of outputting the information which explains the format syntax and looping through the command table to display the commands. Since the command table contains more than 40 formats, the display is split up into two parts, control point command display and the system command display. One may toggle between the two by depressing the equals key.

DOC is the subroutine coded to format and display an entry in the command format table. The algorithm is unpack the format into YBUF, one character per byte, right justified, and then interpret, format and output the appropriate information on a character by character basis.

One should note that this display need not be updated when adding new commands unless one adds a format that allows a string of more than 10 letters, or letters or numbers.

The Z Display

The Z display is the simplest of the displays. The only thing that should be explicitly mentioned is that one should remember to update the description table when adding a new display.

6.10 Some Notes on the Macros

Macros are used rather extensively in DSD to group together common and related sequences of operations, to perform dirty work for the programmer and to bring out any underlying structures in the code. Most of the macros are rather short and some of them and the techniques involved will be described here.

PPMAC is a general collection of macros that are basic to PPU programming. The LDK and ADK macros are basic to most of the collection. The LDK macro will generate a LDC, LDN or LCN instruction depending on the value of its argument, which may be an address expression. The cases are as follows: if one of the elements of the argument is not defined then we generate and LDC instruction, otherwise we force COMPASS to evaluate the address expression by the statement X SET A {remember that COMPASS does straight text substitution of its arguments} so that statements of the form LDC ZR will be handled properly. If the argument is negative and |X| 100B we generate LCN |X|. If the argument is positive and X 100B we generate LDN X. In all other cases an LDC instruction is generated. The ADK macro is similar, except that if |X|=0 then we do not generate any code. One should also note that in the case the argument is negative we perform the following operation:

X SET A and X SET -X.

To see why this is necessary consider the evaluation of the macro

```
     ADK   -10+5
X    SET    A yields X= -5
X    SET   -X yields X= 5 as expected.
```

The EENM macro is similar to the ENM in SCPTEXT except that the exit point always has an X appended to it and the symbol RETLOC$ is set to address of the exit line. One may exit from the subroutine by using the RETURN macro which generates a UJK RETLOC$. If the RETURN macro is given a non-blank argument then the symbol RETLOC$ will be reset to the address of the generated jump. This allows the programmer to save space by returning to the entry point via a series of short jumps instead of a long jump.

Other macros that are local to the displays and are of interest are:

XC	generate VFD for an X coordinate
YC	generate VFD for a Y coordinate
OUTC	output a constant {LDK,OAN}
OUTWN	output N bytes {LDK,OAM}
CNB	call CNB subroutine to clear n bytes
CCS	call CCS subroutine
CALLMTR	call SMF to send a MTR function
INO2	convert 2 octal digits and display them

IN04	convert 4 octal digits and display them
02	call subroutine 02
04	call subroutine 04

6.11 Resident DSD. Subroutines

Main Loop

DOS	Display One Screen
MDC	Monitor Display channel
AVC	Adjust Display Cycle Times
PDR	Process Dayfile Requests
LSH	Left Screen Header Display
KBD	Keyboard Display
CPOMD	Control Point Zero Message Display
RSH	Right SCreen Header Display

System Interaction and Utility

SMF	Send Mtr Function
DCH	Drop channel
RPP	Request a PPU (M.RPP)
CCP	Change Control Point
CNB	Clear N Bytes
RCH	Request a Channel
DFC	Dayfile a Command

Resident Display Subroutines

DOL	display one line
02	display 2 octal digits
04	display 4 octal digits
05	display 5 octal digits
D504	display 5 groups of 4 octal digits
Z2	display 2 octal digits with zero suppression
Z2S	display 2 octal digits with zero suppression
Z6	display 6 octal digits with zero suppression
CCS	change charcter size
A10	display word in BCD
DLB	display linear buffer (one character/byte)
DIT	display internal text ('DIS' format)

Keyboard Input Processing

PKI	Process Keyboard Input
SBT	Search Binary Table
UPC	Unpack Character from format string
CKE	Check Keyboard Entry for a match against command format table entry

SCOPE

The following are "subroutines" of CKE

CCPN	check for control point number
COD	check for octal digits
CLET	check for letters
CLOD	check for letters or digits
CCI	check character interval
CFS	check finite set
SST	set scan termination

Command Execution

LOV	overlay loader {similar to R.OVL}
LDOV	Load and Link display overlay
EXC	Command Execution Control Routine
CPP	Call a PP to an unoccupied C.P. or C.P.O.
SCB	Set the clear bit
CDR	Convert octal digits
ACL	Assemble characters left justified
ASN	Assemble Name left justified.

Command Exits

EXIT.NN	normal command exit
EXIT.FE	FORMAT ERROR MESSAGE exit
EXIT.IE	ILLEGAL ENTRY MESSAGE
EXIT.EE	general ERROR EXIT
	LDC ERROR MSG LJM EXIT.EE
EXIT.DFC	exit for CP Commands that are to be dayfiled
EXIT.DFS	exit for System Commands that are to be dayfiled

6.12 Tables in DSD {in order of appearance}

The following is a list of tables in DSD and a short description of the Table.

For information pertaining to this format of the table one should consult a current listing.

SDTBL	screen display address table
EQKTBL	='s key table
CMDTBL	command format and address table
ODTBL	octal digit conversion table
FCT	first character table
SCT	special character table
RBSTBL	right blank display change table
TEF	command address table extention flag 0 no extention, 1=extention to left screen display, etc.
LST,RST	left and right screen display names
CATFD	holds address of command address jump table for commands residing in an overlay {TEF≠0}
DBATBL	display buffer address table
DOTBL	display to overlay table
RCTBL	resident command address table
ICTBL	table of installation defined illegal channels {in8D1}

6.13 Locations of Interest

LS,RS	left and right screen display address
* DSEQA	address of display eqpt entry
* TCST	holds T.CST
PMKA	address of +- key routine
HP.FCT	address of first character table of higher priority
STPTBL	set to 62B when in step mode, else 0
KRBI	right blank display change index
* CDGLPN	holds an LPN X instruction where X= machine size -1/10000B i.e. X = 7 for a 32K machine
HPRA	holds address of H display parameter change routine address
CMDCA	holds address of routine to change the C display parameters.
* SEC	holds L.RQS/2-N.DEVICE i.e the number of slots in the request stack
* TRQS	holds T.RQS/2 + N.DEVICE

* Set at deadstart time.

6.14 The Macros in PPMAC

On the following pages is a description of the macros used in DSD and DIS. The first part is a description of the macros in the COMDECK PPMAC. One should note that it is a general macro collection for PPU programming and the macros in section four are not used by the display drivers or they are later redefined in DSD and DIS.

The macros are listed in order of their appearance in DSD, and one who wishes to understand them should use the documentation as an aid to the supplement information in the listing.

Macros in PPMAC

LDK ADREXPR *

General LDK macro. Does LDC, LDN or LCN depending on the value of its argument, which may be any valid address expression.

ADK ADREXPR

General ADK macro. Generates an ADC, ADN, SBN or no code depending on the value of its argument, which may be an address expression. If any symbol in the address expression is undefined then the macro will generate an ADC instruction.

SBK ADREXPR

General macro to generate and ADK -{ADREXPR}. All symbols in ADREXPR must be defined.

RAL LOC *

Macro to generate an RAD LOC or RAM LOC depending on the value of LOC. If LOC is not yet defined then it will generate an RAM LOC.

STORE LOC *

Stores the lower 12 bits of the A register in LOC.

LOAD LOC *

Sets the value of the A register to the contents of LOC. That is, it geneates an LDD LOC or an LDM LOC depending on the value of LOC.

PLUS LOC *

Adds the contents of LOC to the A register. That is, it generates an ADD LOC or an ADM LOC.

SUB LOC *

Subtracts the contents of LOC from the A register. Generates an SBD LOC or SBM LOC.

EENM NAME

Generates the entry and exit lines for a subroutine.

---

* If the argument is not yet defined then the appropriate C or M type instruction will be generated.

Example

```
        EENM   ABC generates

ABCX    LJM    0
ABC     EQU    *-1
RETLCO$ SET    ABCX
```

RETURN S

Generates code to exit from a subroutine whose entry/exit ine was formed by using the EENM macro. If the S parameter is present then the subroutine exit point will be reset to the address of the generated code

For example the statement

```
          RETURN S generates

ASYMBOL   EQU  *
          UJK   RETLOC$
RETLOC$   SET   ASYMBOL
```

while the statement
RETURN generates
UJK   RETLOC$

CALL   NAME,LOC

Calls subroutine NAME after loading the contents of LOC in the A register i.e. LOAD LOC followed by RJM NAME

CALLC   NAME,ADREXPR

Generates an LDK   ADREXPR followed by an RJM NAME .

SETC ADREXPR,LOC

Generates an LDK ADREXPR followed by a STORE LOC

MOVE   LOC1,LOC2

Moves the contents of LOC1 to LOC2.

AMB LOC1,LOC2

Forms the difference {LOC1} - {LOC2} in the A register.

RAC   ADREXPR,LOC

Generates an LDK ADREXPR , RAL LOC
For example: RAC -10D,YC will generate
   LCN -10D, RAD YC

LRAL LOC1,LOC2

Generates an LOAD LOC1 followed by RAL LOC2

LAT T,LOC,ADR,INS

Load and test location. Genetates an
LOAD LOC
INS if present
TJN ADR
For example LAT Z,CM,XYZ,{LPN 1}
generates Ldd CM
LPN 1
ZJN XYZ

CLEAR DCELL or CLEAR {DCELL1,.....,DCELL6}

Use to clear {zero} a bunch of direct cells that are not contigious in storage.

For example CLEAR {CM,CM+4} generates
LDN 0
STD CM
STD CM+4

GEN INS,LOC1,... ,LOC6

A recursive macro that generates INS LOCi for i=1,...,6
The number of arguments may be between 1 and 6.

The following macros in PPMAC are not used by the display drivers, or are redefined before they are used. They are ment to be used by programs executed in the Pool PPU's

CALLMTR FUNC
Generates LDN M.FUNC
RJM R.MTR
For example CALLMTR DPP generates
LDN M.DPP
RJM R.MTR

DAYFM MSGADR,FLAGS

Used to call R.DFM to generate a dayfile message. FLAGS is a number between 0 and 77B or may be absent. MSGADR is the address of a message in display code.

The macro generates

LDC MSGADR+FLAGS*10000B
RJM R.DFM

```
    RCH   NUM
    DCH   NUM
```

These two macros are used to request and drop a channel.

If the symbol CH.NUM is defined then the macros will generate

```
    LDN   CH.NUM
    RJM   R.RCH or R.DCH
```

else they will generate

```
    LDD   NUM
    RJM   R.RCH
```

For example DCH FST will expand to

```
    LDN   CH.FST
    RJM   R.DCH
```

OVERLAY NAM,LA

This macro may be used to load an overlay in one of the pool PPU's
The code generated is

```
LDC    3R → MNA
STD   D.T6           store
SHN   -6            name of overlay
STD   D.T7
```

followed by

```
LDC    LA
RJM    R.OVL
```

if the load address is given, else

```
LJM   R.OVLJ
```

FCPA LOC,SAVE

This macro extracts the control point number from LOC, forms the C.P. address in the A register and stores it in the direct cell SAVE if the macro is given a first argument.

FCA   LOC,M

This macro forms an 18 bit address in the A register which is assumed to be stored right justified in LOC-1 and LOC The second argument should be present if the upper 6 bits of LOC are known to be zero.

FCAL   LOC,C

This macro forms an 18 bit address in the A register which is assumed to be stored left justified in LOC-1 and LOC. The second argument should be none zero if the lower 6 bits of LOC may be nonzero.

SCA24 LOC

Saves the contents of the A register right justified in LOC and LOC+1

SCA42 LOC

Save the contents of the A register in LOC and LOC+1 in the format 12/AB,12/C if {A}=ABC originally

RSC42 LOC

The opposite of SCA42

BKN ARG

A macro for auto breaking of a PPU program that is to be debugged under the control of 1DP.

READD CMA,LOC
WRITED CMA,LOC

Macros to read and write from LOC in PP memory to a central memory address CMA.
They generate

```
LDK     CMA
CRD     LOC    or    CWD    LOC
```

READI LOC1,LOC2 and WRITEI LOC1,LOC2

These macros generate

```
LOAD     LOC
CRD      LOC2    or CWD   LOC2
```

READAP A,ADREXPR,PPMEM

Generates

```
LOAD   A        ommitted if A is not present
ADK    ADREXPR
CRD    PPMEM
```

The same goes for WRITEAP.

The following macros are local to DSD and DIS

ADDR A,B

Generates a VFD 12/A, and VFD 12/B if the second argument is present.

OUTW LOC

Outputs the contents of LOC i.e. LOAD LOC, OAN CH

OUTWN N,LOC

Output N bytes starting at LOC
i.e. LDK N, OAM LOC,CH

OUTC ADREXPR

Output a constant
LDK ADREXPR, OAN CH

IPC

Input a character from the keyboard to the A register
FCN 7020B,CH function channel for input
AN CH,IAN CH, DCN Ch

CPAF ORD,CPN

Control Point Address function
CPN is the name of a direct cell holding the control point number, ORD is the name of a W point symbol in the C.P. area.

```
CPAF    CPSTAT,DCPN  generates
LDD     DCPN
SHN     7
ADK     W.CPSTAT
```

LDPP NAM

Used to load the name of a PPU routine {ABC} in the A register in the format CAB

LTDFB

Forms the value of T.DFB in the A register

RDLP COUNT

Used to form the prologue for a short count down loop
Generates

```
        SETC  COUNT,RF
RDL.S   SET   *
```

RDLT

Generates the terminating code for a loop started by RDLP.
Generates

```
SOD  RF
NJN  RDL.S
```

XC N

This macro forms an X coordinate for column N as a VFD statement.

YC N

Forms a VFD for a Y coordinate on line N of the display.

OUTXC N

Generates code to output on X coordinate for Column N.

OUTYC N

Generates code to output a Y coordinate for line N.

CNB FWA,WC

Generates code to call the subroutine CNB to clear WC bytes starting at FWA. The code generated by the macro is:

```
LDK   FWA+WC*10000B
RJM   CNB
```

If the second argument is ommitted then the subroutine will clear 5 bytes.

The following macros are used to generate code for the overlay prologue and termination.

OVERLAY NAM,ORIGIN

This macro forms a segment call for an overlay with name NAM origined at ORGIN. It also saves the start of the overlay in ORG. and the third character of the name in the micro OVLID.

ENDO

This macro is used to total up the size of the overlay and increase the value of BUFSIZE if necessary.

COMMANDS N

This macro generates the prologue for a command overlay. N is a number between 0 and 9. The number 0 is for the resident commands. Actions taken by this macro are to zero the number of commands in the overlay CMDINDEX, call the OVERLAY macro if N ≠ 0 to form a segment and to generate the first two bytes of the overlay which are of the form VFD 12/I.8D→N, 12/IL→N. The symbol I.8DN points to the jump table at the end of the overlay. The second byte holds the name of the overlay and is used by the overlay loader LOV. The macro also initializes the micro CMDTBL to the empty micro.

ENDC

This macro is used to end a command overlay. First code for a jump table is formed by calling the GENTAB macro then the overlay size is determined by calling the ENDO macro.

GENTAB M,MIC

This macro generates an address table from the values of the symbols in the micro string MIC. MIC is assummed to consist of a string of 8 character symbols "invented" by COMPASS. M is the number of symbols in the string. The macro extracts the symbols from MIC and forms a VFD 12/SYM for each symbol extracted.

LNKSYM COMMAND.

This macro generates the information necessary to link up the code for a command with its entry point in the command table. It is discussed in the section on adding commands

LNKSYM CMDTBL {SYNTAX}

This macro is used to generate entries in command syntax table. It is discussed in the section on adding commands.

PROTECT

This macro should be placed before entries in the command format table that the installation wishes to protect. The effect of the macro is to set the protected mode switch for the command.

IN02 TEMP

This macro generate the code to convert two octal digits {right justified in the A register} to display code and output them. TEMP is the name of a direct cell that the generated code stores into.

IN04 LOC

Generates the code to convert 4 octal digits to display code. The source of the digits is the direct cell LOC or if no argument is given then the lower 12 bits of the A register. The macro uses T1 and T2 as temporaries.

D2 LOC
D4 LOC

Call subroutine D2 or D4 after loading the contents of LOC in the A register if it is specified.. The subroutines D2 and D4 will convert octal digits to display code and output them.

CCS N

This macro is used to call the subroutine CCS to change the character size. N may be 0, 1 or 2 for small, medium or large character size.

The following macros are used in the display overlays.

DISOVL N

This macro is placed at the beginning of a display overlay. It generates the information necessary to initialize the overlay. This consists of a SEGMENT card origining the overlay at 77B, a pointer to the relocation table {VFD 12/I.8DN-77B} and the name of the overlay, VFD 12/ILN.

The macro also initializes the number of relocated commands NRAE. and the micro holding the generated symbols.

DEND

This macro is placed at the end of a display overlay. It calls the macro GENTAB to build a relocation table, terminates it with a zero byte and calls the ENDO macro to compute the size of the overlay.

DISHDT N,EKT,PMKA,INIA

This macro generates information that links the display overlay to resident DSD. One of these should appear right after the DISOVL card for each display that will be in the overlay.
In addition to generating the symbol .DIS→N which is used in the DOTBL to tell DSD what overlay the display is in it generates a 4 byte table with the following information:

```
VFD  12/IR→N       or 0 if EKT is missing
VFD  12/PMKA-77B   or 0 if PMKA no specified
VFD  12/DSNN-77B   ordinal to the subroutine entry point
VFD  12/INIA-77B   or 0 if no initialization routine is speci-
                   fied.
```

LOC OVBUF SIZE

This macro is used instead of a BSS statement to generate a buffer in a display overlay. Before generating a BSSZ for the buffer it checks the size of the overlay against the size of previously assembled overlays. If the total size of this overlay is smaller then it doesn't assemble a BSSZ.

It also checks the size of the buffer against the size of the relocation table. If the size of the buffer is smaller than the size of the relocation table, then their is no need to allocate seperate storage for the buffer.

REDEF OPCODE

This macro is used to define macros which redefine the M type instructions in the display overlays so that they put out relocation information when necessary.

ARAT TBL,SYM

This macro is used to generate a symbol whose value is the present value of the position counter -100B and add it to the micro TBL and increment the value of SYM by 1. It is called by the redefined M type instructions when they have to add a symbol to the relocation table.

The macros LDCR, ADCR,LPCR and LMCR are the redefined versions of the C type instructions. They should be used when one wishes to reference a symbol in a display overlay.

OBXXB XX
OBBXX XX

Two macros that used in the BandK displays to output ΔΔXXΔΔ and ΔΔXX respectively.

6.15 DIS

DIS is a display program that may be brought to an empty control point to initiate programs or to an occupied control point to monitor the progress of job. The main features of DIS are that while it is attached to a control point the automatic advance of control cards is stopped and that DIS will not drop out when the error flag is set unless it is set to ▽OPERATOR DROP▽ or ▽KILL▽, hence the programmer is protected from losing the job if he enters invalid control statements, the CPU program being debugged makes an out of range address reference, etc. The system headers and displays are similar to DSD's, but in general they are orientated to debugging of CPU program. Similar remarks hold for the commands.

GENERAL CONSIDERATIONS

Like DSD, one is cramped for space in DIS, but one overlaid display driver was enough for the author; DIS was kept simple. For the interested, it should be noted that DIS was rewritten after DSD, and most of the code in DIS was borrowed from DSD. In general both the logic and the code is simple. Some of the problems that we encountered in writing DIS were:

a) The channel table problem.

Since one may have more than one 6612 per configuration, whenever DIS gets a channel from MTR it must mask the channel number into all the channel instructions. A brief glance at the reference map will show that the normal method of using an address table to insert the channel numbers would occupy approximately 240B bytes. This is far too much. Instead of the aforementioned, we use the ▽shotgun▽ method to insert channel numbers (subroutine GDC). The algorithm is to search for instructions of the form 70XXB where XX = the number of the last assigned channel and insert the present channel assignment.

b) System Transactions

Like an other PPU program DIS must occasionally issue MTR functions. To this it has its own subroutine SMF, which is equivalent to R.MTR in PP resident, except that after a few milliseconds it does a RJM to the main loop (similar to DSD) only to return after one cycle. The purpose of this is to prevent from losing the display in case of a hung MTR, etc. At present this breaks down

because DIS calls R.PAUSE in PP resident.

Another problem that arises, is the programmer asking DIS to excute a statement before all (PPU) activity remaining from the previous run has subsided. In this case DIS drops the CPU and waits for all other PP's at the control point to drop. This causes problems when a wayward PP is hung at the control point (something that should only happen during debugging).

Like DSD, DIS contains similar code in the main loop to handle incomplete system transactions with the same entry point name and flag.

c) Keyboard Entry processing in DIS is not interpretative.

The main reason for this is a lack of space. Other reasons against it are the small number of commands, which are not fixed syntax so an interpretator would have very little work to do. Other things that were done were to allow the operator to omit redundant punctuation. That is ▽DROP.▽ becomes ▽DROP▽ and ▽ENFL50000▽ and ▽ENFL,50000.▽ are equivalent entries.

d) The initialization code in DIS occupies over 300B locations.

After initialization, this space is occupied by the keyboard buffer, the octal digit table and the saved exchange package.

The following pages we will discuss subroutines and sections of code of interest.

INITIALIZATION - starting at INIT

Here DIS sets up the direct cells and various pointers. It then checks the control point to see if it occupied. If it is, and it has non-zero priority, then it proceeds to secondary initialization. If the control point is initially empty then it sets the job name, requests priority, an infinite time limit (77777B), sets the clear bit and requests storage. The algorithm here is to request the minimum of UAS-2 and IP.DSFL/100B; where UAS is the unassigned CM/100B; if at least 300B words of CM is available. After getting a non-zero FL, DIS clears the first 100B words of memory and gets a job sequence number. Note that in the case of a "occupied" control point with zero priority DIS drops out. Control points in the class are JANUS and NEXT control points and it makes no sense to bring up DIS at them.

Secondary initialization consists of getting a display assigned to the C.P. getting the channel, clearing the keyboard buffer and setting up the octal digit table.

We then jump to the main loop.

The subroutines in DIS are:

DOS — Display One Screen
Similar to DSD with a few minor simplifications.

GDC — Get Display Channel
Gets a display channel from MTR and modifies the channel instructions if necessary.

DOL — Display One line
Identical to the subroutine DSD as are D2,D4,D5D4, z2,Z6,Z2S,CCS,D5D4 and A10.

D5 — Convert 5 digits to display code.
A faster version of the one in DSD because the COMPASS mnemonic display is slow.

LSH — A simplified version of the LSN in DSD.

KBD — Keyboard display - Again is a somewhat simpler version.

DLB,DIT — Identical to DSD

RSH — Right Screen Header Display
Almost identical to RSH in DSD.

DSAA — The A display
The same as the code in DSD except that its always the displays the dayfile buffer of the control point that it is attached to. Since the control point dayfile buffers are never larger than 100B we can

only display the last 9 messages. Given all this empty space it decided to take the F {file} display from DSD and place it on the bottom of the screen.

DSBB The B display.
This is similar to the B and K displays inDSD with a display of the exchange package, the last value of the error flag and the break point address thrown in.

DSCC-DSGG These are the central memory displays. Again the code is similar to that in DSD. The only new feature is the companion COMPASS mnemonics with the F and G displays. The two subroutines involved are DINS and display instructions and DOOC-display one op code. The two subroutines use a straight forward, but slow algorithm, hence when the F and G display are both up the screens will flicker considerably

Special Key Processing

PSK Process star {asterisk} key
If the channel that DIS has is DSD's channel, then it will drop the channel and wait in a loop until it gets it back. Note that it holds onto the display equipment so that if one hits the * key in DSD the same DIS will get the channel back.

P=K Process ='s Key - a simplified version of the code in DSD.

RBDC Right Blank Display Change - ditto

PPMK Plus-Minus Key processing

CRK Process Carriage Return

CEB Clear Key processing

BSK Backspace Key processing

IEXC Incomplete Command execution

PKI Process Keyboard Input
A simplified version of PKI in DSD.
All we do here is input the character, check it for validity {< 63B and the first character must be <56B}, search the first or special character tables and add the character to the string if it did not find a match in any of those tables.

CKE Check Keyboard Entry
This routine is called from CRK and RMK to interpretate and process the keyboard entry when the operator hits the carriage return or repeat mode key. The code is as follows: if the first character

is a letter then we assume a memory entry and process it, else we search the command format table for a match of a leading string of letters {e.g. ▽ENFL▽ or ▽X.▽}. If we find one there we jump to the appropriate command processor to execute the command. If we don't find a match in the command format table, then we check for an entry of the form XX, and assume it is a command of the format C4,nnn,etc. If the third character in the buffer is a period or zero {end of entry}, then we assume it is a display change command and process it.
If all of the above has failed then we check for a PP call. One may use DIS to call any PPU program whose name begins with a letter to the control point. The key board entry NAM,XXXX becomes the call

vfd-18/3LNAM,6/C.P. 18/0,18/XXXX

and the entry NAM,XXX,YYY or NAM{XXX,YYY} etc is translated to VFD 18/3LNAM,6/C.P,18/XXX,18/YYY.

Command Processing Routines

None of these routines will be discussed in detail. Most of the code in them consists of assembling strings of octal digits or character strings and checking for syntax errors, etc. One should compare this code with the code in DSD to see what an interpretative keyboard processor buys one. On entry to most of the command processors KP holds a pointer to the address of the first unchecked character in the keyboard entry.

The names of some of the processors and subroutines are listed below:

ENM	process memory entry
PDC	process display change
PMKCDG	+- key processing
CDGMDC	processing for entries of the form Cn,nnnn
CCSB	clear the control statement buffer
XCS	transmit control card to the control card buffer.
CPA	check peripheral activity {wait for all quiet}
ADS	assemble delimated string called to assemble strings of digits of the form ▽ ,nnnn,▽ , ▽nnnn,▽. ▽ , nnnn.▽ etc.
RBW	Restore Breakpoint
DCPU	Calls MTR to drop the CPU
AEE	assemble Exchange Entry i.e. one of the form ▽i,nnn▽.
CDR	convert detail digits right justified

ACL assemble characters left justified

ASN assemble name {left justified}

SMF send MTR function

CNB Clear N Bytes

CPS Check Program Status
Resets RA and FL, checks the error flag, drops the PP if set to ▽OP DROP▽ or ▽KILL▽ else it clears the error flag and exits.

DROPD Drops the equipment and display channel if DIS has them.

PAUSE Pauses for a possible storage move

HOLD Drop the display equipment and channel and stay at the C.P.

CPP Calls a pp program to the control point

MBKP Monitor Breakpoint
checks for P=breakpoint address

Use of the Direct Cells

In general the use of the direct cells is similar to that of DSD. The cells T1-T7 CM-CM+4 and AB-AB+4 are used as tempories and conventions regarding the subroutines in DSD also holds true in DIS. The other direct cells are:

CPRA control point RA/100B } set in CPS

CPFL control point FL/100B } set in CPS

RCP control point that a display is relative to set in DOS

CC cycle count - set in the main loop

XC and X coordinate of 0, read only

YC temporary used as a Y coordinate by the display subroutines

ON constant one

RF repeat flag - used by the display subroutines etc.,

KS pointer tothe start of the keyboard buffer

KI pointer to the first empty byte in the keyboard buffer

KP pointer to the first unmatched character on entry to the command processors if we found a match in the command keyword table.

KML address of error message such as ▽FORMAT ERROR▽ etc.

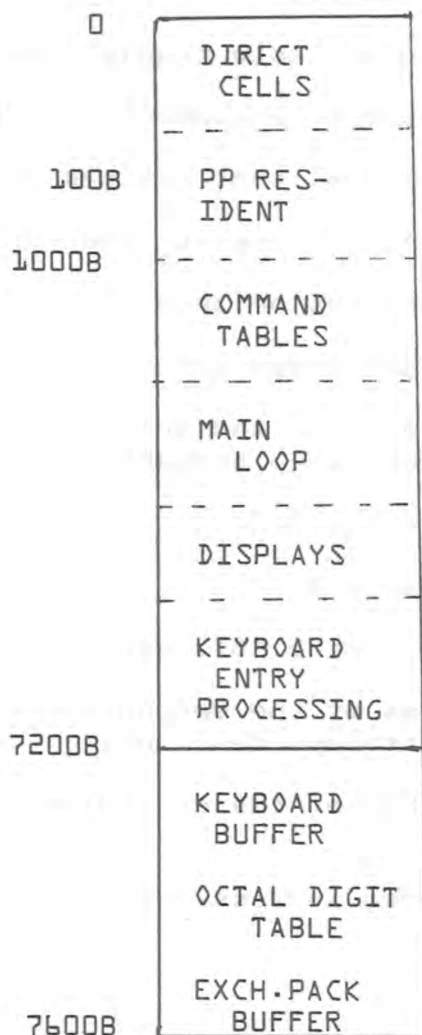
IST	Incomplete System Transaction Flag
PPCR	PP call register - 5 direct cells for holding PP calls. Used by CPP.
WS	Wait system flag. Holds the address of messages over as ▽WAIT-PP ACTIVITY▽ etc.
DFBA	Holds T.DFB/10B, read only
DA	Holds ▽OUT▽ for the dayfile buffer.
DY	A display Y coordinate
KRBI	left screen display table toggle index
EQP	number of the display equipment that DIS has.
D.CH	number of the channel that DIS has or 0 if none.
DCPN	control point that we are at, read only.
BKPA	address of breakpoint or zero.
BKPW	contents of breakpointed word if a breakpoint is out.
LEF	Last non-zero value of the error flag. Cleared whenever one executes a statement.
SI	Screen Index used by DOS.
TEST	Holds T.EST, read only
LEST	Holds the LWA+1 of the EST, read only
DFMZ2	Holds the address of the second dayfile message line on the B display, read only.
FCSP	Pointer to the FWA of the control card buffer, read only.
CPAA	Control point area address, read only

The Reference Map

In DIS the RJM op code is redefined in the beginning the listing so as to produce a "subroutine call (or RJM) sub reference" map. This map is located at the end of the reference map and has the form

.NAME NUMBER OR CALLS REFERENCES

This reference map was used when DIS was written to eliminate some subroutiens that only called once or twice, etc.



6.16 9DM

9DM is a PPU routine, loaded at C.PPFWA, that is called by DSD to:

a. load in display or command overlays that reside on mass storage (are not central memory resident);
b. send a message to the dayfile for DSD.

In the first case, byte 2 of the input register is non-zero and contains the third letter of the name of the overlay to be loaded. The first two letters are "8D". The PP will be attached to C.P.0. 9DM calls the resident routine R.OVL to load the overlay. Upon return from R.OVL it clears byte two of its input register to tell DSD that it has the overlay. It then goes into a counted loop in which it periodically checks byte two of its input register. If it is non-zero then it transmits the overlay to DSD across the display channel (10B). If at the end of 120 milliseconds DSD has not responded, then it drops the PPU and exits to the idle loop.

The reason for 9DM is that DSD (or any other program) can not cancel a stack request. If a direct DSD--1SP overlay loading scheme was implemented, then the operator would be locked out of the keyboard when he pressed the carriage return key and pressing the clear key would not terminate execution of the command.

When 9DM is called by DSD to dayfile the command, byte 2 of its input register is 0. The lower six bits of byte 3 contain flags that are to be passed to R.DFM and locations 164B--171B contain the text of message left justified in "CM DIS" format. 9DM is attached to the appropriate control point. 9DM reads up the message, calls R.DFM to enter the message in the dayfile and drops the PPU.

Because 9DM is an exceptionally short routine (<100B bytes long) it should be CM resident.

6.17 1MH--Mother's Helper

1MH is a PPU routine, loaded at C.PPFWA, which executes the following commands for DSD:
n. X LFNAME,FL., n.UNLOADuu. and N.DEVADDuu. and EVICT, LFNAME,T.
The parameters are passed to 1MH in bytes 2-4 of its input register. In particular byte two contains the index into a jump table in 1MH.

The code in 1MH is simple and straight forward. It consists of loading up the input register, determining the

number of the command to be executed and jumping to the appropriate command processor. It then executes the command and drops the PPU. One should note that their is no facility for 1MH to send an error message to DSD. Any checking for conditions that would make execution of the command "illegal" should be done in DSD.

CHAPTER 7 - TABLE OF CONTENTS

7.0	CIO - 4ES - 6WM	7-1
7.1	CIO	7-1
	7.1.1 General	7-1
	7.1.2 Entry Information	7-1
	7.1.3 Functions	7-2
7.2	4ES	7-8
	7.2.1 Subroutines	7-11
7.3	6WM	7-13
	7.3.1 General	7-13
	7.3.2 Functions	7-13
	7.3.3 Entry Information	7-13
	7.3.4 Exit Information	7-13
	7.3.5 Remarks	7-14

CHAPTER 7 TABLE OF CONTENTS

7.0	CIO - 4ES - 6WM	7-[illegible]
7.1	CIO	7-1
	7.1.1 General	7-1
	7.1.2 Entry Information	7-1
	7.1.3 Functions	7-2
7.2	4ES	7-8
	7.2.1 Subroutine	7-11
7.3	6WM	7-13
	7.3.1 General	7-13
	7.3.2 Functions	7-[illegible]
	7.3.3 Entry Information	7-[illegible]
	7.3.[illegible] Exit Information	7-[illegible]
	7.3.5 Remarks	7-[illegible]

7.0 CIO - 4ES - 6WM

7.1 CIO

7.1.1 General

CIO will perform general functions which apply to all I/O requests. The primary functions of this routine are to validate the existence of the file (or create one if it doesn't exist), check the buffer parameters, validate subsequent requests for I/O in a logical order, and load the proper device driver to perform the function.

7.1.2 ENTRY INFORMATION

a) PP Input register contents for CIO

0-17	FET ADDRESS
18-35	SKIP COUNT
36-38	Control point number
39	Not used
40	Auto Recall
41	Internal Open/Close
42-59	Name CIO

b) FET--The FET will contain the current function code request for this file.

	BITS		
FET (1)	0	Busy bit 0=busy 1=not busy	
	1	Mode bit 0=coded 1=binary	
	2-8	SCOPE CIO codes 0-177	
	9-13	error flags	
	14-17	record levels	
	18-59	file name display code	
FET (2)	0-17	buffer parameter (first)	
	18-23	(length of FET)-5	
	24-35	disposition code	
	36-43	not used	
	44	EP bit, user processing error	
	45	UP bit, user processing EOI	
	46	release RB's	
	47	random bit	
	48-49	Tape density	for tapes;allo-cation style for mass storage
	50-51	label	
	52-53	data type	
	54-59	device type	

	BITS	
FET (3)	0-17	buffer parameter (in)
FET (4)	0-17	buffer parameter (out)
FET (5)	0-17	buffer parameter (limit)
	18-32	PRU size
	33-47	RB size
	48-59	FNT Pointer

7.1.3 Functions

a) Validate the FET

checks first and last location to see if FET is within field length. If not, check EP bit and return to user or abort with dayfile message.

b) Compare FET and FNT file names

1) If the FNT POINTER in FET is not equal to zero:
   a) Check the pointer to see if it points to an entry within the table.
   b) Retrieve entry and compare

2) If the FNT POINTER in FET is equal to zero:
   a) search for empty entry and file name
   b) if none exists, create an FNT into empty entry

c) Update FET, FNT

1) Set FET, FNT to busy.
2) Transfer function code to FNT and save last code status in FNT.

d) Check buffer parameters

1) LIMIT $\leq$ FL
   OUT $<$ LIMIT
   IN $<$ LIMIT
   OUT $\geq$ FIRST
   IN $\geq$ FIRST

e) Verify the CIO function code

1) Any function as defined by the SCOPE CIO CODES is legal. If not defined, EP bit checked. Error flagged in last code status, FNT, FET. Control returned to user or error message to dayfile and abort

2) Check the sequence of operations this file except for random files.
   a) read after write sequential file
   b) skip forward after write sequential file
   c) if last function was re-write in place above is ignored.

f) Security checking

See Table A

g) Overlay Loading

Overlays will be loaded in the fillowing manner, the prefix from the overlay name (XAA). X specifies the locations at which loading will begin. For X=1, the loading will begin at 1000B.

X-2,,,7 loading will be at
X&1000B

h) Exit Information

1) Magnetic Tapes

For read and write functions control is transferred to the driver with the file at busy status.
For skipf and skipb control is returned, CIO sets file status to complete and drops out.

2) Allocatable Devices

CIO loads 4ES and either 4ES or 1SP can complete the request depending on the function code.

3) The direct cells are set by CIO as indicated by Table B.

i) Errors detected by CIO

Number	Remarks	Owncode No.
1	FET OUTSIDE FL	22
2	CIO CODE NOT DEFINED ON DEVICE	22
3	ILLEGAL FILE NAME	32
4	READ OR SKIPF AFTER WRITE	30
5	FILE NOT OPEN FOR WRITE	22
6	WAITING FOR FNT SPACE	24

Number	Remarks	Owncode No.
7	FILE NOT OPEN FOR READ (PERM.FILES ONLY)	22
8	BUFFER ARGUMENT ERROR	31
9	ERROR CONDITION NOT CLEARED LAST REQUEST	22
32	READ PERMISSION NOT SET FOR READ ON P.F.	22

j) Overlays called by CIO

1)	1RS	Read stranger tape
2)	1RT	Read internal SCOPE 3 tape
3)	1MT	Read/Write other tapes
4)	1WX	Write external tapes
5)	1WS	Write stranger tapes
6)	1WI	Write internal tapes
7)	2PC	On-line card punch
8)	2RC	On-line card reader
9)	2LP	On-line printer
10)	6WM	Write error message
11)	4ES	Mass storage I/O

	FUNCTION	NO FNT	CLOSED LABELED	CLOSED UNLABELED	OPEN LABELED SEC READ	OPEN LABELED SEC WRITE	OPEN UNLABELED SEC READ	OPEN UNLABELED SEC WRITE	RANDOM OPEN	RANDOM CLOSE	PERMANENT OPEN	PERMANENT CLOSE
TAPES	READ	CIO creates an FNT for alloc device	Error	Error	CIO tape driver	CIO tape driver	CIO tape driver	CIO tape driver				
TAPES	WRITE	CIO creates FNT for alloc device	Error	Error	Error	CIO tape driver	Error	CIO tape driver				
	READ	CIO creates FNT for alloc device calls 4ES	Error	Error			CIO 4ES	CIO 4ES	CIO 4ES	Error	CIO 4ES	CIO IMS 4ES
	WRITE	Same	N/A	Error	N/A	N/A	Error	CIO 4ES	CIO 4ES	Error	CIO 4ES	CIO 4ES

TABLE A

TABLE B

Direct Cell Settings by CIO

$CELL_8$	SYMBOL		REMARKS
0-15	D.Z0-D.T5	Scratch	
16	D.T6		
17	D.T7		
20	D.FNT	FST{1}	6 BITS EQP TYPE BITS ALLOC OR LABEL BITS
21	+1		first RBT word pair or 2nd unit ord
22	+2		current RBT or primary unit ord
23	+3		current RBT ord or current PR count
24	+4		current PR or current PR count
25	+5	FST{2}	6 BITS unused 6 BITS FET address
26	+6		low 12 bits FET address
27	+7		disposition code
30	+8		4 access bits 2 security bits 6 code status
31	+9		code status
32	D.EST	EST	assignment
33	+1		channel assignment 1, 2
34	+2		channel assignment 3, 4
35	+3		hardware mnemonic
36	+4		DST ordinal
37	D.DTS		allocation from FET device type from FNT
40	D.BA	FET{1}	first two chars file name
41	+1		second two chars file name
42	+2		third two chars file name
43	+3		last char filename, rec level 4, error 2
44	+4		error flags 3, code status 9

CELL$_8$	SYMBOL		REMARKS
45	BS	FST{2}	code and status of previous operation
46	D.FR6		not used
47	D.FR7		not used
50	D.PPIRB	PPIR	overlay name CI
51	+1		overlay name O, Int bit, recall bit, CP numb
52	+2		skip count
53	+3		skip count 5, FET address 6
54	+4		FET address
55	D.RA		relative address of control point
56	D.FL		field length of job
57	D.FA	FST	FST{1} address in CM
60	D.FIRST	FET{2}	buffer parameter first
61	+1		buffer parameter first
62	D.IN	FET{3}	buffer parameter in
63	+1		buffer parameter in
64	D.OUT	FET{4}	buffer parameter out
65	+1		buffer parameter out
66	D.LIMIT	FET{5}	buffer parameter limit
67	+1		buffer parameter limit
70	D.PPONE		contains 1
71	+1		1 bit random bit FET, 11 bits length FET
72	+2		
73	D.TR		
74	D.CPAD		address of control point area
75	D.PPIR	PP	address of PP input register
76	D.PPOR	PP	address of PP output register
77	D.PPMES1	PP	address of PP message buffer

7.2 4ES ENTER STACK REQUEST

This overlay may be called to prepare stack entries for all file content dependent I/O commands for files residing on allocatable storage devices, and to perform directly some logical manipulation of such files.

4ES is called with a part of the results of a call to CIO or 2BP. The expected contents of the direct cells are as follows:

D.CPAD 0 for request at control point zero. Non-zero otherwise.

D.PPIRB+2 - D.PPIRB+4
Normal parameters to CIO, i.e., record count n {where relevant}, and FET address {when present}.

D.FNT - D.FNT+9
the FST entry for the file.

D.FA the CM address of the FST entry

D.FIRST - D.FIRST+1
D.LIMIT - D.LIMIT+1
These values are placed in the stack request without any checks.

D.EST contains flags for the stack request. These may include a PP available flag {bit position 0} and a no FET flag {BIT S.STFETP - S.STFA} either or both of these bits may be set in this flag, but no others should be set by the calling program.

D.EST+1 contains a CP recall request flag
= 0 for no recall at end of request.
= 1 for recall at end of request.
This flag is ignored if D.CPAD=0.

If the FET flag in D.EST is 0:
D.BA contains FET word 0.
Otherwise in the case of the open function {D.FNT+9 = 100b}

D.BA contains FET word 2 {only the random bit is significant}.

The last previous status of the file is in BS {cell 45B}. 4ES may be called with the following function codes in D.FNT+9. Note that the two low order bits are always ignored

READ {10B}. READSKIP, {20B}, WRITE {14B}
WRITE RECORD {24B}, WRITE EOF {34B}, BACKSPACE {40B}, REWIND/ UNLOAD {50B, 54B, 60B, 64B.} OPEN {100B}, SKIP FORWARD {240B}, SKIP BACKWARD {640B}, BACKSPACE PRU {44B}, EVICT 114B}.
REWRITE {214B}, RE-WRITER {224B}, REWRITEF {234B}, READNS {250B}

Initial procedures in 4ES include..

1. Stuffing several parameters into a ten word area called REQUEST. These are the flags from D.EST and D.EST+1,

FIRST and LIMIT taken from the low core cells D.FIRST and D.LIMIT, and FET address taken from D.PPIRB {low order 18 bits}. the FST location taken from D.FA. The order and record level field of the request are cleared at this point as well.

2. The location/100B of the RBT {LWA+1} is put in D.Z6.

3. The constant 2 is put in D.Z7. Neither D.Z6 nor D.Z7 are altered by 4ES thereafter.

4. If there is an RBT chain for the file, its first word is read into D.Z1 - D.Z5. Otherwise these cells are initialized to zero.

5. The last buffer code, taken from BS and the current request code taken from d = CS+1 are saved. The two low order bits are stripped of these values, thus no attention is given to the BCD/binary bit or the ready bit of either value.

The first major decision of 4ES is whether or not the request which it is treating is random. This is determined from the FET by the joint condition 1 ⁺2 random bit set, request/return field non-zero. In the no FET case the request FET by the joint condition 1=2 random bit set, request/random after clearing the request/return field. Note that a file need not be opened as a random file in order to be driven with random requests.

RANDOM resets the local value of the last buffer code, since a RANDOM read or write legitimately follows any request, to prevent special action in later processing. It then branches to read or write actions. The random read action checks for release in progress or requested, as this is not proper in conjuction with random I/O. The file is positioned, using INDEX {SCANRBT is used for EDITLIB}, to the point specified in the FET request/return field and the FST is reset to this position. Processing continues at ES3 where normal read action is handled.

The random write action also checks for release in progress. If so it evicts the current RBT and joins new file action which established a new RBT of random type. In any case the file is repositioned to EOI. Return information is now constructed, consisting of the PRU ordinal counted from the start of the RBT chain assigned to the file. The return is only made if requested. Normal wirte action is rejoined following the checks peculiar to sequential write.

AT ES2 sequential I/O branches to read action ES3, or write.

In write the file is evicted in case releasing is in progress and new file action establishes an RBT of sequential type. If new file action is not necessary, and the file is of random type, control goes to RANDOM. Otherwise, the current position of the file is tested. If it is not an EOI all RB's beyond the current position are released. Now all writes rejoin at ES25 where a test is made for write EOR{34B} command. If so, and the last command was not 24B or the buffer is not empty, the EOR flag is added to the stack request. Otherwise for EOR the record level is set to 17B in the stack request. For all write commands the EOI PRU is reset in the RBT in case it may have been altered due to write before EOI. Control then branches to COMPLETE.

At ES3 normal read action branches to one of several actions. These are..

SKIPF	n is added to REQUEST and tested for -0. If n=-0 the file is positioned to EIO and control transferred to EXITR. Otherwise processing continues at READSKP- the level number, taken from D.FNT+* is added to REQUEST. Processing continues at READ - test are made for release in progress or requested, last operation write or file at or past EOI. Otherwise processing continues at COMPLETE.
BKSPRU	n is added to REQUEST as is the order O.BPRU.
BKSP	n = 1 is added to request as is the order O.SKPB.
SKIPB	n is added to REQUEST as are the level number taken from D.FNT+8, and the order O.SKPB. These three join at BACK, where a test is made for release in progress. If so control transfers to EVICT. Otherwise the file position is tested for BOI. In that case the BOI exit is taken {status - 1}. Otherwise control goes to COMPLETE after the point where order is added to REQUEST.
OPEN	creates a new RBT if one does not already exist. This is random or sequential as the random bit in FET word 2 is 1 or 0. Device type and allocation are extracted and stored in D.DTS for the calling routine. Control goes to EXIT.
REWIND	current file position is reset to beginning of file. If there is an FET, IN = OUT = FIRST is set in CM. At EXITR a check is made for release in progress. If not control goes to exit. Otherwise control goes to.

EVICT — the file's RBT is dropped. Control goes to EXIT.

Error — Status is set to 22B and the FET and the FST made ready by RSTCS. If the error bit is on control goes to EXITER, past the call to PSTCS. Otherwise the proper dayfile message is issued and the control point aborted.

At EXIT code and status are reset in the FNT and the FET (if any) with the ready bit on. If recall was requested monitor function M.RCLCP is called. If the available bit was set the PP is dropped and return goes to the idle loop. Otherwise a normal return is taken.

At COMPLETE the proper order is added to REQUEST. The unit number is determined from the RBT and associated RBR and added to REQUEST. R.EREQS is called to enter REQUEST in the stack. If R.EREQS returns, a normal return is taken.

7.2.1 Subroutines of 4ES

TFL — test for a value within control point field length. A switch directs return to ERROR or the alternate return from FETA in case of range error. If no error TFL returns the absolute address of the value in A.

FETA — adds the FET address taken from D.PPIRB+3, +4 to a value in the accomolator and checks its range by a call to TFL. If there is no FET, the alternate return from FETA is with 0 in A.

RSTCS — called with the new status in a, this routine resets code and status in the FET and the FNT with ready bit equal to 1.

SETN — extracts n from D.PPIRB+2, +3 and stores it in REQUEST. If n = +0, control goes to EXIT (i.e., the command is a pass except for random read positioning).

RELCHN — constructs a stack request to release a chain of RB's whose header address is given in D.T1.

DRPRBT — calls RELCHN to drop the entire RBT of a file. This subroutine is essentially EVICT. Allocation is saved in the PRU field of the FET.

GETRBR — chooses an RBR for a new RBT.

STRTRBT — presets a new RBT. Allocation type 1 is specified for random files if not specified

by the D.DTS or by the PRU field of the FST.

RSTFST	writes the FST in CM.
GETRBT	fetches an empty RBT. If the empty chain is empty it calls monitor to assign more space and creates another 40B empty RBT word pairs.
SCANRBT	scans forward in the RBT until reaching a specified RB number /PRU or EOI. Leaves the found RB address in d = RBTA { entered only when Absolute Index Flag in FET is set}
FINDRBR	computes the address of an RBR table and reads its first word.
SETADC	stores the address computed by FINDRBR. in routine ADC.
ADC	adds the address to A.
CHECK	communicate with the operator when a file cannot be assigned to a device, because it is full or unloaded.
DECIDE	For ECS, checks for special named files and files with non-zero disposition code. These files are not assigned to ECS.
PERM	For ECS and RE-WRITE orders RE-WRITE sets jump instructions to PSN's. PERMANENT FILES checks file operation for: 1. Release bit illegal for permanent file 2. Evict function not allowed 3. Write allowed on files if extend permission set and file at EOI. RE-WRITE allowed if modify permission set and not at EOI.
INDEX	converts the specified PRU ordinal to RBT/RB/PRU address, or gets address and PRU ordinal for EOI PRU.
NEWRBR	gets RB length for the specified RBR according to the device type or special allocation type.

7.3 6WM MESSAGE OVERLAY

7.3.1 General

6WM was written to issue dayfile messages and insert own-code error numbers into the FET and FNT for illegal input/output requests.

7.3.2 Functions

1. Check the EP bit in FET{2}-44 if set issue no dayfile messages but insert owncode error no's in FET, FNT drop PP.

2. If EP bit not set issue dayfile messages, insert own code error numbers in FET, FNT abort PP.

3. The FET, FNTwill be set to complete status prior to dropping the PP.

4. For an FNT full condition the display message WAITING FOR FNT SPACE is posted and CIO is requested as a delayed job with a 2 second wait. Bit 41 is set so that CIO will clear the message if an FNT empty entry is available. If not the operator may drop the job and normal EP processing will be handled by 6WM.

7.3.3 Entry Information

Direct Calls

21	D.FNT+1 contain the error number in two positions. Bits 0-5 primary error number 6-11 secondary error number
40-43	D.BA-D.BA+3 file name in display code 7 characters miximum
53-54	D.PPIRB+{3,4} the FET address

This routine is entered with an RJM instruction.

7.3.4 Exit Information

1. The dayfile message appears as follows:

```
ILLEGAL I/O REQUEST
FILE NAME XXXXXXX
FET ADDRESS XXXXXX
```

One of {N} reasons for error secondary message if required.

2. Bits 9-13 of code and status FET and FNT will contain an owncode error number as described in the reference manual.

3. The FET and FNT are set to complete status.

Addition of errors.

1. Add the message to table MSG

2. Add corresponding owncode number to CPCT

3. Add message address to MTBL

7.3.5 Remarks

The bit used to communicate with CIO (bit 41 D.PPIRB+2) is used by OPE and CLO as the internal call bit so that if CIO is used as a focal point for open and close functions also, some other communication link will have to be used.

CHAPTER 8 - TABLE OF CONTENTS

8.0 JOB PROCESSING 8-1

8.1 1RA - RESOURCE ALLOCATOR 8-2

- 8.1.1 Entry / Exit Conditions 8-2
- 8.1.2 General Description 8-2

8.2 1AJ - JOB ADVANCEMENT 8-5

- 8.2.1 Design Philosophy 8-5
- 8.2.2 Logic Flow 8-5
- 8.2.3 Major Subroutines 8-5
- 8.2.4 Control Card Processing 8-7
- 8.2.5 Environment 8-9
- 8.2.6 Other Routines Using 1AJ 8-9
- 8.2.7 System Tables Used 8-9
- 8.2.8 Memory Usage 8-9
- 8.2.9 Usage of Routine 8-10
- 8.2.10 Residence of Routine 8-10
- 8.2.11 Special Assumptions 8-10

8.3 1EJ - END-OF-JOB-PROCESSOR 8-11

- 8.3.1 Philosophy 8-11
- 8.3.2 General Description 8-11
- 8.3.3 Major Section of Code 8-14
- 8.3.4 Other Routines Called 8-15
- 8.3.5 Memory Usage 8-15
- 8.3.6 Usage Routine 8-15
- 8.3.7 Residence 8-15
- 8.3.8 Equipment Interfaces 8-15
- 8.3.9 Stack Requests - Timing 8-15

8.4 1LT - LOAD JOB FROM TAPE 8-17

- 8.4.1 Purpose 8-17
- 8.4.2 General 8-17
- 8.4.3 Method 8-17
- 8.4.4 Notes 8-19

CHAPTER 8 - TABLE OF CONTENTS

8.0 JOB PROCESSING 8-1

8.1 1RA - RESOURCE ALLOCATOR 8-2

- 8.1.1 Entry / Exit Conditions 8-2
- 8.1.2 General Description 8-2

8.2 1AJ - JOB ADVANCEMENT 8-5

- 8.2.1 Design Philosophy 8-5
- 8.2.2 Logic Flow 8-5
- 8.2.3 Major Subroutines 8-5
- 8.2.4 Control Card Processing 8-7
- 8.2.5 Environment 8-9
- 8.2.6 Other Routines Using 1AJ 8-9
- 8.2.7 System Tables Used 8-9
- 8.2.8 Memory Usage 8-9
- 8.2.9 Usage of Routine 8-10
- 8.2.10 Residence of Routine 8-10
- 8.2.11 Special Assumptions 8-10

8.3 1EJ - END-OF-JOB-PROCESSOR 8-11

- 8.3.1 Philosophy 8-11
- 8.3.2 General Description 8-11
- 8.3.3 Major Section of Code 8-14
- 8.3.4 Other Routines Called 8-15
- 8.3.5 Memory Usage 8-15
- 8.3.6 Usage Routine 8-15
- 8.3.7 Residence 8-15
- 8.3.8 Equipment Interface 8-15
- 8.3.9 Stack Requests - Timing 8-15

8.4 1LT - LOAD JOB FROM TAPE 8-17

- 8.4.1 Purpose 8-17
- 8.4.2 General 8-17
- 8.4.3 Method 8-17
- 8.4.4 Notes 8-19

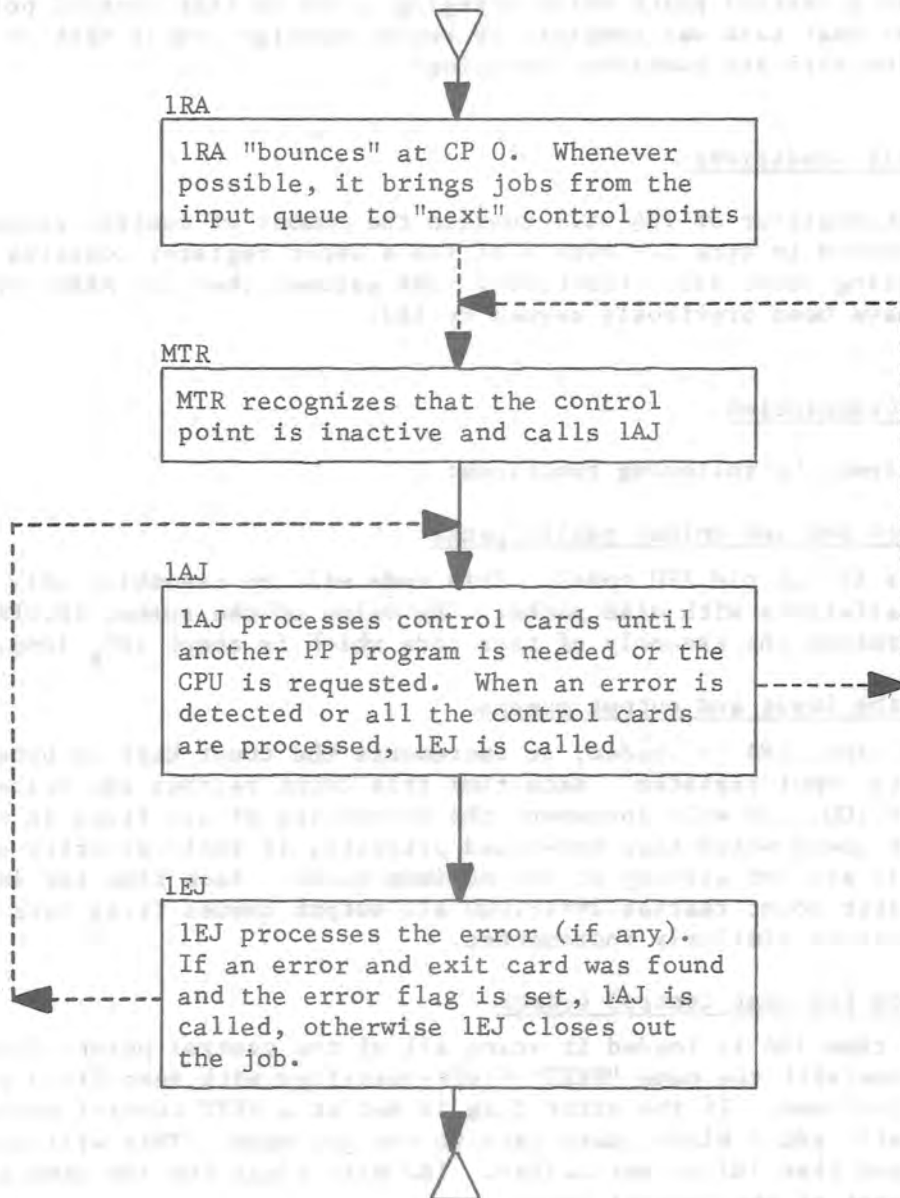
8.0 JOB PROCESSING

Job processing, as described here, deals with that part of the SCOPE operating system pertaining to job scheduling, startup, advancement, and close.

Job processing spans three major PP codes. Each code performs the following distinct functions:

a. 1RA Scheduling and job startup (Resource Allocator)
b. 1AJ Job advancement (control card processing)
c. 1EJ Job close and error processing

The following flow chart describes the general flow and communications between routines.



8.1 Resource Allocator - 1RA

1RA has been written for SCOPE 3.1.6 to perform functions related to bringing jobs in the input queue to control points. These functions were previously performed by 1BJ. In addition, 1RA will include all the code which previously was in the routine 1FU to free public disk packs when they are inactive. 1RA is approximately 1000_8 bytes long and should always be central memory resident. Putting 1RA on disk could tie up stack processor. At deadstart time, MTR would contain an entry for 1RA in the PP job queue. Thereafter, each time 1RA is done performing its function, prior to dropping, it would insert itself into the PP delay stack with a delay of 250 milliseconds. The internal symbol "DELAY" in 1RA determines this delay. The effect would be to have 1RA bouncing at control point zero looking for "NEXT" control points and available jobs in the input queue. 1RA would attach itself to a control point while bringing a job to that control point but after that task was complete it would reassign itself back to control point zero and continue "bouncing".

8.1.1 Entry/Exit Conditions

The input register of 1RA will contain the number of control points in the system in byte 3. Byte 4 of 1RA's input register contains the queue ageing count described below. 1RA assumes that all NEXT control points have been previously zeroed by 1EJ.

8.1.2 General Description

1RA performs the following functions:

1. Search for and unload public packs

   (This is the old 1FU code). This code will be assembled only at installations with disk packs. The value of the symbol IP.DPAK determines the assembly of this code which is about 100_8 long.

2. Age the input and output queues

   Each time, 1RA is loaded, it increments the count kept in byte 4 of its input register. Each time this count reaches the value 2**IP.IQD, 1RA will increment the priorities of all files in the input queue which have non-fixed priority, if their priority sub-levels are not already at the maximum value. Each time the input register count reaches 2**IP.OQD all output queues files have their priorities similarly incremented.

3. Search for next control points

   Each time 1RA is loaded it scans all of the control points looking for one with the name "NEXT" (left-justified with zero fill) as the job name. If the error flag is set at a NEXT control point, 1RA will add 2 blank characters to the job name. This will cause 1AJ and then 1EJ to get called. 1EJ will clear the job name and the rest of the control point area.

4. Search the queues

If there are no available, NEXT control points and neither queue is to be aged this time, 1RA will put itself back into the delay stack. Otherwise 1RA searches the FNT looking for input queue jobs and incrementing priorities in either queue when appropriate. If either queue is to be aged the FNT channel is reserved for the entire search. 1RA looks at all input queue jobs to find the "best" job to bring to a control point. The "best" job is defined to be the fixed priority job with the highest priority or, if there are no fixed priority jobs, the "best" job will be the job with the highest priority that will fit in the currently available central memory and ECS storage. If two acceptable jobs have equal priority, the one with the larger CM requirement is chosen.

5. Initiate the "best" job at a control point

If 1RA finds an acceptable input queue job and there is at least one available NEXT control point, 1RA will try to initiate the running of the job at the lowest numbered NEXT control point. First 1RA assigns itself to the control point and then requests storage for the job. If the storage request is rejected (either CM or ECS) for a job with non-fixed priority the available storage must have shrunk since 1RA first checked it. In this case 1RA will go back and search for another job. If the storage request was refused and the job had fixed priority, 1RA will not attempt to bring any other jobs to any control points. It will write the message "FIXED PRIORITY JOB XXXXXXX, WAITING FOR STORAGE" (where XXXXXXX = the name of the job) to the "B" display under the selected NEXT control point, and then 1RA will put itself back into the delay stack.

If the requested storage is assigned 1RA does the following:

a. Reserves the FNT channel and checks the input queue FNT of the selected job to ensure that no other routine has picked it up.

b. Just prior to releasing the FNT channel, 1RA rewrites the input file FNT entry so that the file is type local at the control point and the file name is "INPUT". The disposition code is saved (it is checked by RESPOND).

c. Sets the control point area error flag to the appropriate value if the abort flag in the input queue entry was set (job card error, etc.).

d. Zeroes RA and RA+1.

e. Sets up the following words in the control point area:

W.CPSTAT	Field length is set when storage is requested. (Error flag is set if job card error or job job pre-abort.)
W.CPJNAM	The job name is inserted and C.CPNCSP is set to zero.

W.CPPRI	Priority is requested.
W.CPTIML	The time limit is inserted and CPU time is set to zero.
W.CPERT	The FST address of the input file is put in C.CPFST. For EXPORT/IMPORT jobs the E/I bit in C.CPFP is set. For RESPOND jobs the RESPOND and "no rerun" bits in C.CPFP are set.
W.CPJCP	The parameters from the job card are saved.
W.FSTCC	The second word of the input FNT entry is stored.
W.CPLDR	The default values for the loader and map (IP.LDR and IP.MAP) are set in C.CPLDR3.
W.CPRES1	Byte 0 is set to 6000B for RESPOND jobs.
W.CPDFMC	The dayfile message count, IP.MSCT, is inserted into C.CPDFMC.

f. An exit mode of 070000B is set in the exchange packge.

g. The job name is issued as the first dayfile message.

h. If this job is being rerun the message "JOB RERUN" is issued.

i. The FNT is searched for a PRE-OUTPUT file (a file local to control point zero whose job name is the file name. A rerun job or pre-aborted job will have a pre-output file.) If a pre-output file is found it is made the output file for the job.

When the job has been competely set up, 1RA assigns itself back to control point zero and then looks for more NEXT control points and jobs in the same manner as described above.

8.2 JOB ADVANCEMENT - 1AJ

8.2.1 Design Philosophy

1AJ is coded for minimum space except in instances where performance is severely hampered. A bare minimum of logic is placed in 1AJ in order for it to perform its basic function of control card processing. Virtually all logic dealing with error processing an job close is delegated to 1EJ. Logic associated with control point initialization and job start-up is placed in 1RA. If another PP program is needed, it is called into the same PP as 1AJ.

8.2.2 Logic Flow

When MTR finds that CPU status + PP job queue count + delay stack count + stack processor entry count is equal to zero, 1AJ is called to process the next control card. The flow chart on the following page describes the general logic flow.

8.2.3 Major Subroutines

GNELM Get Next Element

GNELM is the subroutine used for dismembering control cards. Upon entry the control card is in BUF through BUF+40_{10}. A pointer to the current character in BUF is maintained in BUFADR. A call to GNELM will result in the next element being placed in the direct cells ELM through ELM+4.

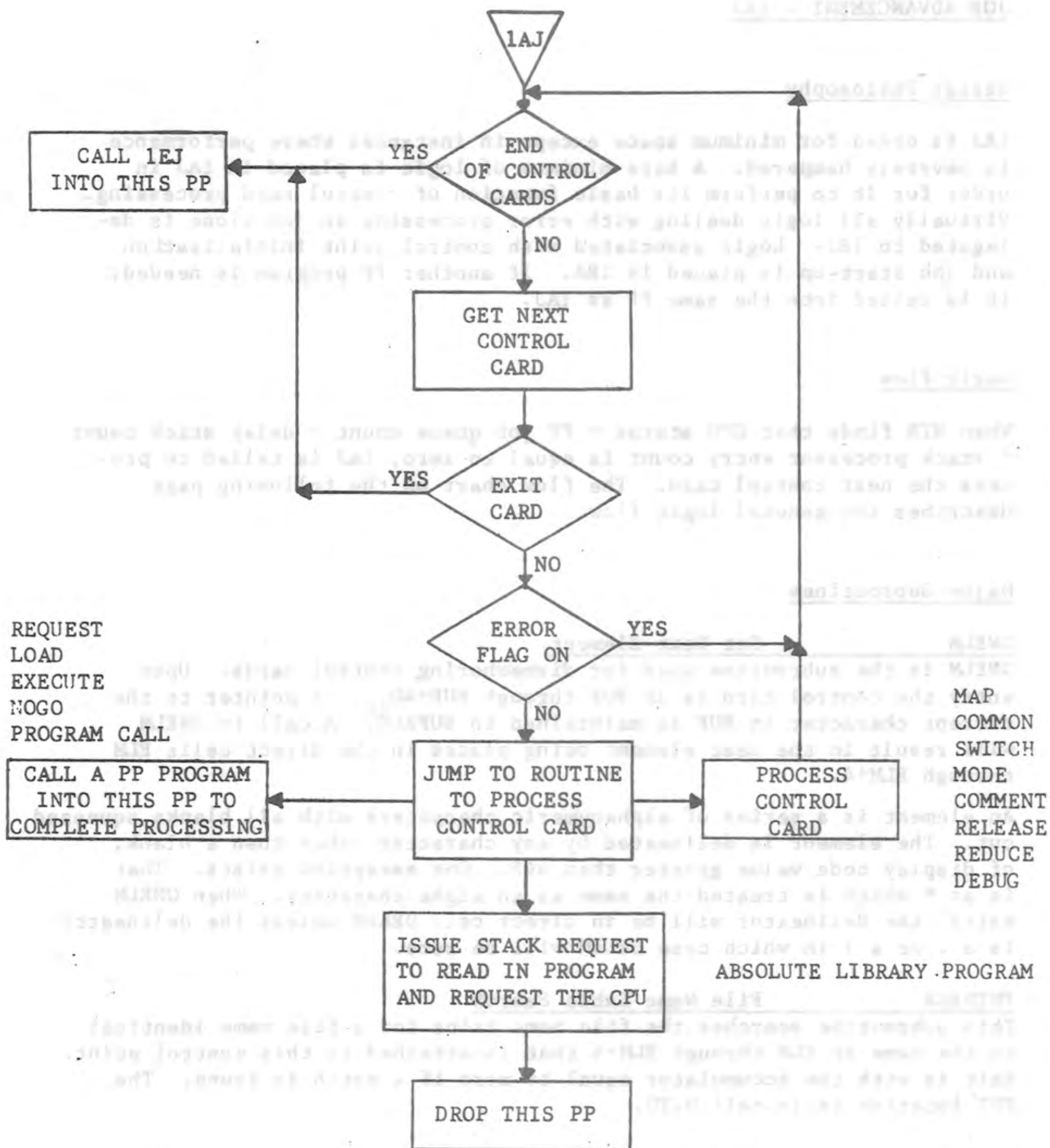
An element is a series of alphanumeric characters with all blanks squeezed out. The element is delineated by any character other than a blank, of display code value greater than 44B. One exception exists. That is an * which is treated the same as an alpha character. When GNELM exits, the delineator will be in direct cell DELIM unless the delineator is a . or a) in which case DELIM will be zero.

FNTSRCH File Name Table Search

This subroutine searches the file name talbe for a file name identical to the name in ELM through ELM+4 that is attached to this control point. Exit is with the accumulator equal to zero if a match is found. The FNT location is in cell D.TO.

PNTSRCH Program Name Table Search

The program name table is searched for an absolute library program with the same name as that contained in ELM through ELM+4. If a match is found, the accumulator will be zero and the two CM words associated with the library routine will be in direct cells WORD1 and WORD2. If a match is not found, the accumulator will be nonzero. PNTSRCH makes use of the fact that EDITLIB groups all absolute library programs at the end of the program name table.


1AJ
END OF CONTROL CARDS
YES
CALL 1EJ INTO THIS PP
NO
GET NEXT CONTROL CARD
EXIT CARD
YES
NO
ERROR FLAG ON
YES
NO
REQUEST
LOAD
EXECUTE
NOGO
PROGRAM CALL
CALL A PP PROGRAM INTO THIS PP TO COMPLETE PROCESSING
JUMP TO ROUTINE TO PROCESS CONTROL CARD
PROCESS CONTROL CARD
MAP
COMMON
SWITCH
MODE
COMMENT
RELEASE
REDUCE
DEBUG
ISSUE STACK REQUEST TO READ IN PROGRAM AND REQUEST THE CPU
ABSOLUTE LIBRARY PROGRAM
DROP THIS PP

RCC Read Control Card PRU

The control card FST entry from the control card area is swappend into the input file FNT entry. A stack request for a READP is issued and the data is then moved into the buffer in the control point area. A short PRU has a zero word added to signal the end of control card.

8.2.4 Control Card Processing

COMMENT	The comment card less "COMMENT" is issued as a dayfile message. The card is issued in two parts if the message is over 40 characters long.
RELEASE (lfn)	The file name lfn is checked to see if it is common, local to this control point and not "SYSTEM". If the tests are true, the file change bit is set.
MAP	The loader control bits in byte C.CPLDR3 of the control area are set to 0 if OFF is specified, 1 if ON is specified or 2 if PART is specified.
MODE	The monitor function M.REM is issued to set the exit mode to that specified on the MODE card. If the mode is greater than 7, a control card error message is issued.
LOAD (lfn,p(1), p(2)...p(n))	The "load" switch is set. The arguments are entered into the field length beginning at RA+2 and continuing to RA+n+1 where n is the number of parameters. RA+64B contains the number of parameters. LOD is loaded into the same PP and control is transferred to LOD.
EXECUTE (name,p(1), p(2),...p(n))	The "execute" switch is set and the operation is the same as LOAD after the "load" switch is set.
NOGO	The "nogo" flag is set and the operation is the same as LOAD after the "load" switch is set.
COMMON (lfn)	The FNT is searched for a file of the same name as lfn. If neither a local nor common file of lfn were found, the message "WAITING FOR COMMON FILE" is displayed and the card is re-processed. If only a common file is found assigned to control point zero, the file is assigned to this control point and processing continues. However, if the common file found has a control point assigned, the message "WAITING FOR COMMON FILE" is displayed and the card is re-processed. If the common file is on a non-allocatable equipment and the equipment is unavailable the message "WAITING FOR COMMON EQUIPMENT XX" (where XX = EST ordinal) is issued and the card is re-processed. If only a local file of name LFN is found in the FNT, the change bit in FST is set and processing continues. If both a local and a common file of name lfn was found, the message "DUP COMMON FILES OF LFN" is issued and the job is terminated.

SWITCH (N)	The parameter (N) sets a bit in the sense switch word (W.SSW) of the control point. N CP sense switch word bit 1 5 2 6 3 7 4 8 5 9 6 10
file (p(1), p(2)...p(n))	For this control card "file" should be the name of a program such as RUN. The "p's" are parameters for this program. The file name table is searched for "file". If "file" is not found, the central memory absolute overlay portion of the library program name table is searched. If "file" is found in the file name table and not found in the program name table, the load switch is set to zero and the operation is the same as that for LOAD after the "load" switch is set. If "file" is a central memory absolute overlay, a small start-up program is written into the user's field length, and a stack request is issued to load the absolute overlay into RA+100. 1AJ then releases the PP. As soon as the stack request is completed the stack processor requests the CPU. If the overlay is CM resident, 1AJ loads the overlay and requests the CM. The start-up program checks the validity of the overlay. If invalid, an error message is issued and the job is aborted.
REDUCE	The reduce bit is set in byte C.CPLDR3 of the control point area.
REQUEST	The parameters are seperated into RA+2 and on, where they will be found by REQ. The number of parameters is left in RA+64B.
CATALOG ATTACH EXTEND PURGE	The permanent file control cards are placed into RA+70B. No dayfile is issued. A central program PFCCP is then loaded to process the cards.
RPACK	The first parameters, the pack name, is placed into D.BA, the second and third, VRN and lable type, into D.FIRST and D.FNT, then 5DA is called.
REMOVE	The first parameter, lfn, is placed in D.BA. If present, the second, pack name, is left in D.EST. 5DA is then called to process the REMOVE function.
DEBUG (C,S,T)	The parameters may be in any order. Loader control bits are set in byte C.CPLDR3 of word W.CPLDR in the control point area.

EXIT(p) — 1EJ is called into the PP. The exit bit in byte C.CPFP of word W.CPERT of the control point area is set depending on the value of the error flag and the setting of the job abort bit in byte C.CPFP. The abort bit is set if a control card error is found or if the binary output of an assembly or compilation which had fatal errors is loaded. If the exit card contained a parameter (i.e. EXIT(S)) then the exit flag is set if the error flag is set. If the exit card did not contain any parameters the exit flag is set only if the error flag is set and the abort bit is not set.

8.2.5 Environment

Other routines used:

a. LOD is called for LOAD, EXECUTE, NOGO or program control cards.

b. REQ is called to process equipment requests (REQUEST card).

c. 1EJ is called when all control cards have been processed or when an EXIT card is found.

d. 5DA is called to process private pack control cards.

8.2.6 Other Routines Using 1AJ

a. MTR calls 1AJ for job advancement.

b. 1EJ calls 1AJ when an EXIT card exists and the error flag was set.

8.2.7 System Tables Used

FNT — The file name table is searched in order to find a common file during RELEASE card processing. The FNT is also searched as a protion of the logic to determine if a program call card is calling an absolute library program.

8.2.8 Memory Usage

1AJ occupies about 4500_8 words of PP memory or about 730_8 words of CM memory if it is CM resident.

8.2.9 Usage of Routine

1AJ is sequentially reusable.

8.2.10 Residence of Routine

1AJ is normally CM resident in order to minimize disk accesses during control card processing. If CM space is at a premium, 1AJ could be placed on disk at a cost of a maximum of 1 disk access per control card and 2 disk accesses at the end of a job. Normally, only disk access at the end of the job would occur and somewhat less than 1 disk access per control card is needed due to the fact that in some cases 1AJ can process more than one control card without being reloaded.

8.2.11 Special Assumptions

No more than 41 parameters are allowed on any absolute library program call. This restriction occurs because a small program is loaded into RA+53B that edits level numbers and transfers control to the library program. This technique is employed so 1AJ can drop out before the stack request to load the library program is completed.

8.3. 1EJ - End-of-Job-Processor

The following write-up presents a general description of 1EJ. A more detailed description is included as comments in the code.

8.3.1 Philosophy

1EJ is a combination of several functions which at one time were separate programs. Combining these programs reduces the number of stack requests necessary to load the code for end-of-job processing and eliminates some duplication of code. Where possible, all calls to CIO have been eliminated by making stack requests through PP resident. In addition, no central memory is used for transferring the dayfile to the output file or for dumping the exchange package. This allows 1EJ to release storage sooner and also eliminates the possibility of hanging up waiting for a couple hundred words of central memory. The dayfile transfer is accomplished by reading from the dayfile to a buffer in 1EJ using READP and then writing to the output using WRITEP. The exchange package is dumped using central reads to a 1EJ buffer and then a WRITEP to the output file.

8.3.2 General Description

1EJ, the end-of-job processor is called by 1AJ in the following cases:

1. When all control cards for the job have been processed.
2. When an EXIT or EXIT(S) card is encountered.
3. When the error flag for the job is set.

In the first two cases the job will be terminated. In the third case the job is terminated except when the error flag is set to a value other than rerun or kill and an EXIT or EXIT(S) card is contained in the job. In this latter case 1EJ will issue a dayfile message describing the error, dump the exchange package and the area from P-100B to P+100B, flush the buffers of files with special names or non-zero disposition and then call 1AJ to process the control cards after the exit card.

In all other cases 1EJ will terminate the job as follows:

1. Dispose of all the files associated with the job by one of the following means:
   a. Drop the file and all equipment or storage associated with the file.
   b. Attach the file to control point 0 for further processing (i.e., common or output files).
   c. Call an overlay to dispose of the file (e.g. 6PC for permanent files.)
2. Transfer the job dayfile to the output file associated with the job.

3. If the error flag is set to something other than rerun or kill, dump the contents of the exchange package and the central memory locations P-100B through P+100B to the output file (DMPX) and issue a dayfile message describing the error.

4. Release storage for the job request, zero priority, set the error flag to zero, set the job name to "NEXT" (or to zeros if the clear bit is set) and zero the control point area from word W.CPTIME to W.CPINS-1.

A. Normal Termination

If the job terminates normally (i.e. with the error flag equal to zero) 1EJ will copy the dayfile to the output file, release storage, reset the control point area and dispose of all the files associated with the control point as follows:

1. Private Disk Pack Files

   All private disk pack files are disposed of before any other files are disposed of. Information about each file (such as name and the RBT chain for the file) is written to the disk pack. The file itself remains on the pack, but the FNT entry for the file is deleted and the FNT entry for the pack is also deleted. RBT chains in central memory for disk pack files are released.

2. Permanent Files

   Each time a permanent file is found, 6PC is called to dispose of it. 6PC will delete the FNT entry for the file but normally the file will still exist on a mass storage device.

3. Common Files*

   The file type in the FNT is set to common; the file is assigned to control point zero. The file name and disposition code are retained. If the file is on a non-allocatable device, the equipment is dropped and turned off. If the file is a random file, and is set to open write or open alter status, the index is written out.

4. Input File

   The input file is not dropped until after the dayfile has been transferred to the output file, and the output file has been released. At this time, the disk space for the input file is released (evicted) and the FNT is zeroed.

---

* 1EJ checks the file type (byte C.FTYPE of the files FNT) and the change bit in the FST to determine whether a file is common or local. If the change bit is not set the file type is that indicated by C.FTYPE. If the change bit is set then the file type is the opposite of that indicated by C.FTYPE (i.e., if C.FTYPE is common the file is local). After the change bit is checked in routine TWODF it is reset.

5. Output File

At the start of 1EJ the output file buffers are flushed (purged). After the dayfile has been copied to the output file, the output file FNT is modified so that the file name is the job name, the file type is output, the file is assigned to control point zero, and the file is rewound. If the file had a disposition code, it is retained otherwise a print disposition is inserted.

6. Other Special Name Files (PUNCH, PUNCHB, FILMPL, etc.)

The file buffers are flushed, and the FNT is rewritten so that the file name is the job name, the file type is local, and the file is assigned to control point zero and is rewound. If the file had a disposition code, it is retained otherwise an appropriate disposition code is inserted.

7. Local Files With Non-Zero Disposition Code

If the file is on a non-allocatable device, the equipment is dropped and the FNT zeroed. This is done because JANUS cannot handle non-allocatable files. If the file is on an allocatable device, the file buffers are flushed, and the FNT is rewritten so that the file name is the job name, the file type is local and the file is assigned to control point zero and is rewound. The disposition code is retained.

8. Other Local Files

If the file is on a non-aalocatable device, the equipment is dropped. If the file is on an allocatable device, the disk space is released. In either case the FNT is zeroed.

9. Dayfile

After the dayfile has been copied to the output file, the dayfile FET is reset. If part of the dayfile is on disk, the disk space is released, the FNT is rewritten to make the file local to control point zero, and the FST is zeroed.

B. Abnormal Termination

The types of abnormal termination and the action taken for each are as follows:

1. KILL

If the error flag is set to 7 meaning that the operator has killed the job, then common files, permanent files, and private disk pack files are processed normally; but all other files (including output and non-zero disposition files) are dropped (i.e., the FNT is zeroed and the disk space released or equipment dropped), a system dayfile message "JOB KILLED" is issued; the control point area is reset and the job name is cleared. There is no output of any kind for this case.

2. RERUN

   If the error flag is set to 10B which indicates that the operator has typed-in "n.RERUN" then the input file is returned to the input queue. Permanent files and private disk pack files are processed as in normal termination. All files which were common at the start of the job are returned to control point zero but all other files are dropped. A pre-output file is created, and the dayfile is copied to this file whose name is changed to the job name and made local with zero disposition to control point zero, and is not rewound. This file will become the output file when the job is rerun. Then before dropping out 1EJ resets the control point area.

3. ERROR

   If the error flag is a non-zero value other than kill or rerun then the job terminated due to some type of error (operator drop is considered to be in this catagory). In this case the error must be processed and the exchange package must be dumped to the output file. Processing then continues as in normal termination including the disposing of files, transferring the dayfile and resetting the control point area.

C. Other Considerations

1. Zero-Priority Jobs

   If the job which terminated has zero priority, 1EJ will set the clear bit on and set the error flag to kill before beginning to process the job.

2. CLEAR

   Normally, when 1EJ resets the control point area the job name is changed to "NEXT". However if the clear bit in byte C.CPFP of the control point area is set, then the job name will be zeroed.

8.3.3 Major Sections of Code (or Subroutines)

ONEJ Start of 1EJ. Calls ERROR to put out dayfile error message if error. Calls PACKS to dispose of private disk pack files.

DOALL Disposes of all files attached to the control point except INPUT, OUTPUT, and private pack files. Calls PURGE to flush file buffers. Calls TWODF for common files or files which are to be dropped. Calls RLS to put files in the output queue.

TWODF Assigns common files to control point zero. Calls RANDOM to write out the index on random common files. Drops all files which are not common.

CLOS Calls XPACK to generate DMPX type dump. Calls DFILE to transfer the dayfile to the output file. Releases the INPUT and OUTPUT files and the DAYFILE. Resets the control point area.

8.3.4 Other Routines Called

6PC is called to dispose of all permanent files.

8.3.5 Memory Usage

1EJ will occupy about 5000 octal locations beginning at location 1000 octal in peripheral processor memory. The length is dependent on the installation parameters IP.RESP, IP.RMT and IP.DPAK. The remainder of memory and a large portion of the code is organized so that it may be overlayed and used for buffer space.

Buffer areas in 1EJ are always accessed using a FIRST, IN, OUT, LIMIT pointer scheme. LIMIT is defined equal to 7777B and is called BUFLIM. The other 3 pointers are stored in direct cells BUFFWA, BUFIN and BUFOUT.

8.3.6 Usage Routine

1EJ is sequentially reusable.

8.3.7 Residence

1EJ may be located in central memory or on an allocatable device. Since 1EJ is called at least once for every job, locating 1EJ on an allocatable device will increase the overhead for a job. In general, this means only one additional access for each job and should be more tolerable than current implementations when central memory space is at a premium.

8.3.8 Equipment Interfaces

1EJ may encounter any hardware device which exists in the system. The philosophy of 1EJ regarding hardware error statuses is not to attempt recovery beyond that provided by the hardware devices. Consider for example, the alternative. If a hardware error occurs which cannot be recovered by the driver, the decision is whether to continue or drop the job. Since 1EJ is in the process of dropping the job, this procedure would serve no useful purpose.

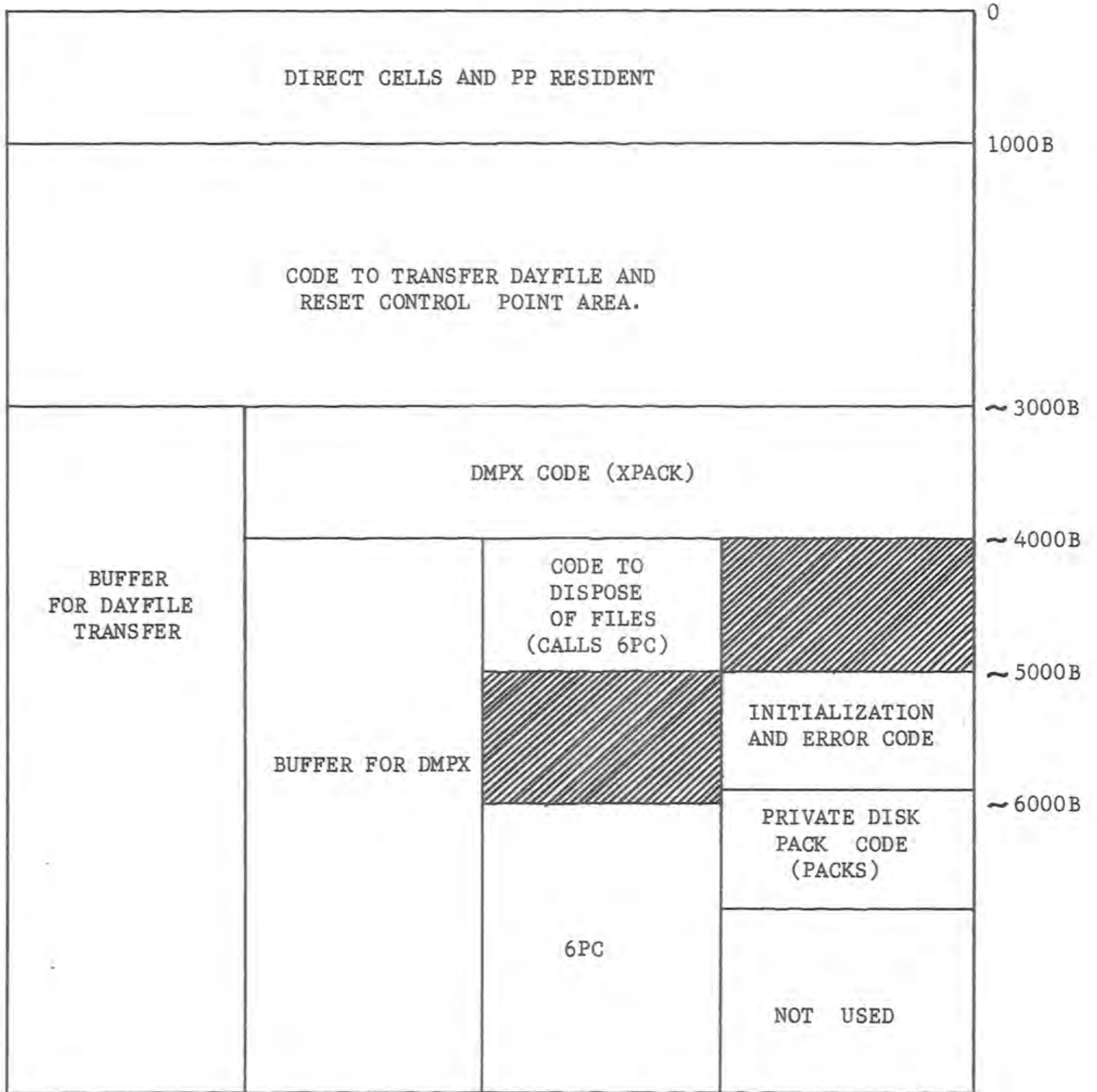
8.3.9 Stack Requests - Timing

All stack requests made by 1EJ are through PP resident (R.EREQS, R.READP or R.WRITEP). At a minimum, 1EJ will issue one stack request to evict the input file and one request to write the dayfile into the output file. If part of the dayfile is on disk, it will take one

or more stack requests to read the dayfile to the PP and a corresponding number of requests will be needed to write the data to the output file. The dayfile buffer can hold about 10 PRU's of dayfile data.

If the exchange package is dumped, the data will fill the exchange package buffer about one and 1/2 times so that an additional 1 or 2 stack requests will be needed to write data to the output file.

MEMORY MAP OF 1EJ



MEMORY MAP OF 1EJ

0

DIRECT CELLS AND PP RESIDENT

1000B

CODE TO TRANSFER DAYFILE AND RESET CONTROL POINT AREA.

3000B

DMPX CODE (XPACK)

4000B

CODE TO DISPOSE OF FILES (CALLS 6PC)

BUFFER FOR DAYFILE TRANSFER

5000B

INITIALIZATION AND ERROR CODE

BUFFER FOR DMPX

6000B

PRIVATE DISK PACK CODE (PACKS)

6PC

NOT USED

8.4 1LT - LOAD JOB FROM TAPE

8.4.1 Purpose

The program 1LT is used to load jobs onto disk via magnetic tape. The usual procedure for calling 1LT is keying through the console keyboard.

8.4.2 General

1LT is called to a control point by DSD when the operator initiates a keyboard entry. The format for the entry is described in the SCOPE Ref. Manual, but briefly the format is Y.LOAD., where Y stands for the control point in question. An alternate format is Y.LOADX., where the X designates the tape to be loaded as a tape not created under SCOPE 3 (external to the system).

A standard circular buffer technique is used to process the input file. The first statement of the first record is presumed to be the JOB card, and it is processed accordingly. This assumption is made for each job on the tape. Standard system I/O is used to both read tape into memory and write memory onto disk CIO, and its associated overlays are called in for this task.

When a stack of jobs have been processed from a tape, and if no more exist, 1LT is released. The end-of-processing indication is a double end of file found on the tape. When this is encountered, control is transferred to the idle loop. The same procedure is followed if an error condition is detected at the control point.

There is no limit to the number of control cards belonging to a given job. Thus, if the user fails to place an end-of-record card behind the job card record, this omission will not be detected at load time.

8.4.3 Method

A. The Main Routine, 1LT

After initial setup, which includes determination of whether tape is internal or external to system, this routine calls EXCP to set up buffer parameters, request storage, etc. If storage is not available, the PP is released and control is transferred to the idle loop. After storage is granted, subroutine RQT is called to issue a request function for tape assignment. Subroutine RQT calls for REQ with a request PP function to assign the correct unit. RQT uses its wait loop until assignment has been made and the request function completed.

A cycle of loading and dumping of information into the buffer from tape, out of the buffer to disk, is entered. Upon encountering an end of file, this process is terminated. A single end of file indicates that end of job has been reached. Then PP time is recorded and the job is released. The process continues until all jobs stacked on the tape have been transferred to disk. The PP is then dropped and control is transferred to the idle loop.

B. Subroutine ENCP - Enter CP and Buffer Parameters

This subroutine tests job name in the control point area. If it is non-zero, the PP is dropped and control is transferred to the idle loop. Otherwise, it places the name LOAD, or LOADX depending on the type of tape specified, into the job name; then it requests storage. If storage is not assigned, it drops the PP and exits to the idle loop. If storage is assigned, it clears RA through RA+2. This creates an FET for the file; it sets FIRST, IN, OUT, LIMIT, and then returns to the main routine.

C. Subroutine RQT - Request Tape

This routine sets up a two word request function parameter list in central memory, requests a PP for REQ (the request program) and then enters a wait routine till the request has been completed. If an error is detected, the control point is aborted.

D. Subroutine DBF - Dump Buffer

This subroutine transfers information from the circular buffer to disk. The FET code and status is set to write out the buffer and the buffer is read in. CIO is called to transfer the information to disk. DBF enters a wait loop until the operation is complete. If an error occurs on the write, the control point is aborted. Otherwise, a normal return is made to the main loop.

E. Subroutine LBF - Load Buffer

This subroutine loads job files from tape into the circular buffer. Initially, the JOB name in the FET is set to tape. The FET code and status is set to read tape. Then CIO is called to transfer data from tape to the circular buffer. Then LBF enters a wait loop until the operation is complete. If an error occurs on the read, the control point is aborted. Otherwise, a normal return is made to the main loop.

F. Subroutine ERR - Error Routine

This routine is called from 1LT following return from 2TJ if a JOB card error has been detected. It assigns default values to field lengths, priority, and time limit. It sets up circular buffer parameters and procedes to complete reading of the job. This reading is done without an accompanied write which is in effect a space forward to end-of-file. Then it returns to the main routine which initiates processing of the next job on the tape.

G. Subroutine REL - Release Job

This subroutine is called upon to release a job after it has been loaded onto disk. The job name is transferred to the FET and 2BP is called to check buffer parameters. Field length (both CM and ECS) and priority are stored in the second FNT word. File status is set to complete. Updated FNT entries are then restored to the File Name Table.

H. Subroutine PPT - Record PP Time

This subroutine records peripheral processor time in the system and job dayfiles.

I. Subroutine SFT - Shift Job Name

This subroutine is used to shift the job name one character to the right before reading it out to the system dayfile.

8.4.4 Notes

A. Routines Called by 1LT

REQ - request function to assign the desired load tape for processing by 1LT.

CIO - handles all tape and disk I/O for 1LT

2TJ - translates job card for 1LT

2BP - creates job file FNT entry for 1LT

B. Dayfile Messages Issued by 1LT

1. During initialization 1LT checks the job name at the assigned control point. If the job name is non-zero, the following message is entered in the system dayfile:

   LOAD(X) .CONTROL POINT IN USE.

2. Following a Pause request to Monitor, the control point error flag is checked. If the error flag is non-zero the following message is entered in the system dayfile:

   JOBNAME. CONTROL POINT ERROR.

3. If a double end-of-file is detected while reading tape, the following message is entered in the system dayfile:

   LOAD(X) .COMPLETE.

4. If an external tape is being used, the FNT is searched and the PRU count is set to zero to prevent rewinding. If the search is unsuccessful, the following message is entered in the dayfile:

   JOBNAME. FILE NAME NOT FOUND.

5. If an error is dected after an I/O operation, the following message is entered in the system dayfile:

   JOBNAME. CONTROL POINT ABORTED.

6. If sufficient storage is not available, the following message is entered in the control point display:

   AWAITING STORAGE

CHAPTER 9 - TABLE OF CONTENTS

9.0 STACK PROCESSOR - SCOPE 3 Mass Storage I/O Package 9-1

9.1 RATIONALE 9-1

9.2 OVERLAY STRUCTURE 9-2

9.2.1 Main Program 1SP and 7SZ 9-2
9.2.2 Driver Overlays 3Sx 9-2
9.2.3 Non-Standard Allocation Overlays 7Sx 9-3

9.3 STACK PROCESSOR INITIATION 9-4

9.4 STACK PROCESSOR COMPLETION 9-4

9.5 ORDER CODES 9-5

9.5.1 Central Memory Read/Write Orders 9-5
9.5.2 Peripheral Processor Read/Write Orders 9-6
9.5.3 Positioning Orders 9-6

9.6 ERROR CODES 9-7

9.6.1 End of Information 9-7
9.6.2 Parity Error 9-8
9.6.3 Invalid RBR Number 9-8
9.6.4 Disk Full 9-8
9.6.5 Buffer Parameter Error 9-8
9.6.6 Not Assembled for ECS 9-9
9.6.7 Undefined Order Code 9-9
9.6.8 Write Impossible 9-9
9.6.9 Invalid RBT Address 9-9
9.6.10 Connect Reject 9-9
9.6.11 RBT Space Needed 9-9

9.7 SYSTEM INTERFACE 9-10

9.7.1 Tables 9-10
9.7.2 Routines 9-11
9.7.3 Monitor Functions 9-12
9.7.4 Common Decks 9-13
9.7.5 Residence Requirements

9.8 PP MEMORY ALLOCATION 9-13

9.8.1 Direct Cells (0000-0077) 9-13
9.8.2 PP Resident (0100-0775) 9-14
9.8.3 Low-core Working Storage (0776-1056) 9-14
9.8.4 1SP Code (1057-approx.6077) 9-14
9.8.5 Middle-core Working Storage (approx.6100-6121) 9-15
9.8.6 3Sx Code (approx.6122-6720) 9-15
9.8.7 7Sx Code (approx.6721-6744) 9-15
9.8.8 Unused Area (approx.6745-7151) 9-15
9.8.9 High-core Working Storage (7152-7777) 9-15

9.9 NARRATIVES 9-17

9.9.1 Initialization 9-18
9.9.2 Search of Request Stack 9-19
9.9.3 Selection of Best Request 9-22

9.9 NARRATIVES (continued)

9.9.4	Initiation of Request Execution	9-23
9.9.5	Request Executives	9-25
9.9.6	Termination of Request Execution	9-32
9.9.7	Advance Record Block Subroutines	9-36
9.9.8	Miscellaneous Subroutines	9-38
9.9.9	Driver Overlays	9-43

9.0. Stack Processor – SCOPE 3 Mass Storage I/O Package

The Stack Processor package, consisting of PP program 1SP and its overlays, performs all operations involving mass storage devices (disks, drums, etc.) for all other components of the operating system (except dead start) and for all programs processed by the system. Any PP program may cause a mass storage I/O operation by using the MTR function M.EREQS (usually by calling one of the PP Resident routines R.EREQS, R.READP, R.WRITEP) to enter a request into a stack of such requests; a CPU program must call a PP program (usually CIO) to issue stack requests for it. Each request contains a device number (DST ordinal), identification of the file (FST or RBT address), and an order code indicating the type of operation (read, write, etc.) to be performed. When a copy of 1SP has finished whatever it was doing, it looks in the request stack for something to do, chooses one of the requests from the stack, and executes that operation. The mass storage I/O package is called Stack Processor because it is driven by a request stack.

9.1. Rationale

In the SCOPE 3 system, mass storage I/O is performed by a specific program rather than by subroutines included in other programs, for the following reasons.

1. Mass storage I/O, including device assignment to files, device space management, and error detection and recovery, requires a large amount of code. The total size of the system is minimized by having only one copy of this code.

2. The various types of mass storage devices supported by the SCOPE 3 system differ greatly in hardware characteristics such as addressing methods, control function codes, and status response byte formats. With the Stack Processor, adding a new type of mass storage device to those that can be supported by the system is achieved by adding a new driver overlay to the Stack Processor package and making minor changes to a few other system components.

3. With most mass storage devices, the time spent in positioning the device between data transfers is greater than the time spent in actually transmitting data. By maintaining a stack of requests for each device, the Stack Processor for a device can choose the request that requires the smallest positioning time from the device's current position, thus minimizing the total amount of time devoted to device positioning. Stack Processor does this, but also uses a priority incrementing scheme to ensure that a request involving a large positioning time will not be bypassed indefinitely in favor of requests with smaller positioning times.

9.2. Overlay Structure

The Stack Processor package is divided into overlays for the sake of modularity and to improve PP memory space utilization. The overlays are grouped into main program, driver overlays, and non-standard allocation overlays. The main program and driver overlays are contained in one assembly (from IDENT to END) with one deck name, 1SP, in the Program Library file; the overlays are separated by SEGMENT pseudo-operations. This facilitates cross-referencing among routines contained in the overlays and eliminates the inefficiencies that result from fixed origins and/or relocatable code.

9.2.1. Main Program 1SP and 7SZ

The main part of Stack Processor is device-independent and consists of the main program 1SP and overlay 7SZ. The overlay contains code that logically belongs in 1SP but does not fit and is seldom used, so it is loaded, when needed, into what is normally the I/O buffer area.

9.2.2. Driver Overlays 3Sx

Each driver overlay contains device-dependent code for one type of mass storage device. This code is concerned with connect, release, position, read, and write operations, error detection and recovery, logical and physical addressing correspondence, and access time calculation for the scheduler.

The code for each driver overlay is contained in a common deck, RMSx, in the Program Library file, because there are two copies of each mass storage driver overlay (except RMSX) in the SCOPE system: one copy, 5Cx, is part of the Dead Start package and the other copy, 3Sx, is part of the Stack Processor package. The driver overlays currently provided are listed below.

common deck	1SP overlay	dead-start overlay	device type	mnem.	code
RMSA	3SP	5CP	6603-I disk	AA	01
RMSB	3SQ	5CQ	6638 disk	AB	02
RMSC	3ST	5CT	6603-II disk	AC	04
RMSD	3SR	5CR	865 drum	AD	12
RMSP	3SS	5CS	854 disk pack	AP	07
RMSX	3SX	——	ECS	AX	20

Dead Start has no driver overlay for ECS. Each common deck name has the same last letter as the corresponding device type mnemonic, while the overlay names follow a different pattern. Since all of the overlays are assembled together with 1SP or CONTROL, possible duplicate names are avoided by having each location symbol begin with the last letter of the common deck name. Thus there are subroutines APOSIT, BPOSIT, ... , XPOSIT, that perform similar functions for different device types. When referring to these routines in a generic sense (i.e., regardless of device type), a period is used to represent the initial letter. Thus .POSIT represents all of the positioning routines.

9.2.3. Non-Standard Allocation Overlays 7Sx

The non-standard allocation (NSA) overlays are used by Stack Processor when working with a file whose allocation style code is greater than 3. Each NSA overlay transforms a non-standard record block (RB) designator into a standard RB designator and a starting PRU number within that standard RB. The name of the NSA overlay used is calculated from the allocation style code as follows:

04-07	7SA	30-33	7SF	54-57	7SK
10-13	7SB	34-37	7SG	60-63	7SL
14-17	7SC	40-43	7SH	64-67	7SM
20-23	7SD	44-47	7SI	70-73	7SN
24-27	7SE	50-53	7SJ	74-77	7SO

Each 7Sx overlay must be relocatable (because it is loaded at different locations for different 3Sx driver overlays) and must be serially reusable (because 1SP will not reload it if it is already in core). At present, the only NSA overlay supplied with the system is 7SB; it deals with the eight-PRU record blocks used by the RESPOND systems.

9.3. Stack Processor Initiation

The System Monitor, MTR, assigns program 1SP to an available PP when any PP issues an enter request stack function, M.EREQS, that references a DST entry in which the active flag (byte 4 of the second word) is zero. MTR also sets the active flag non-zero and places the DST ordinal in byte 4 of the 1SP input register. 1SP is initially assigned to control point zero. When 1SP is loaded into the assigned PP, it looks at the DST entry referenced by its input register and loads a driver overlay 3Sx, where the "x" is a letter in display code found in bits 42-47 of the first word of the DST entry. The DST entry also contains the channel and equipment numbers for the device. This 1SP commences to execute all requests in the stack that reference the same DST entry, ignoring all requests that reference other DST entries. 1SP assigns itself to different control points as necessary, always returning to control point zero whenever a request is completed or reissued to the stack.

9.4. Stack Processor Completion

A Stack Processor releases its PP, and zeros the active flag in its DST entry, when all of the following conditions are satisfied.

1. The DST entry has entry count = exit count; i.e., there are no requests in the stack referencing this DST entry.

2. The PP Job Queue entry count (maintained by MTR in byte 0 of CM word T.MSC for this purpose) is non-zero; i.e., at least one task is waiting for an available PP.

3. The active flags are non-zero in at least two DST entries; i.e., at least one other 1SP is either running in a PP or waiting to be assigned to a PP.

The first two conditions minimize unnecessary dropping-out and reloading of the Stack Processor; the third condition, together with suitable provisions in MTR, ensures that, at all times, at least one PP contains a Stack Processor or one PP is reserved for that purpose. This is required so that it is always possible to load a system program that resides in mass storage.

9.5. Order Codes

This section describes the order codes used in mass storage I/O request stack entries. These codes are different from those used in the CIO code and status fields of FET and FST entries. For each order code, the following descriptions give the octal code, the standard system symbol, the standard system macro and CIO codes if any (note that no distinction is made between coded and binary mode for mass storage I/O), and the actions specified by the order code. The order codes are divided into three groups, corresponding to the three formats for the second word of a request stack entry. Order codes 06 and 07 are not defined at present. No order codes are reserved to installations.

9.5.1. Central Memory Read/Write Orders

00 O.READ Corresponds to the READ system macro, CIO codes 010 and 012. Read data from device to CM until (a) end of information is reached, (b) a short PRU has been read, or (c) the next PRU will not fit into the buffer.

01 O.RDSK Corresponds to the READSKP system macro, CIO codes 020 and 022. Read as for O.READ until (a) or (b) above, or (c) the CM buffer is completely full; then change to O.SKF with N = 1 unless the reading was stopped by (b) with record level ≥ the level in the request.

02 O.RCMPR No corresponding system macro or CIO code. Read as for O.READ but do not transmit the first three CM words of the first PRU. This is used for loading programs from the system library, in which the first three CM words of each record contain information of interest only to EDITLIB and Dead Start.

03 O.RDNS — Corresponds to the READNS system macro, CIO codes 250 and 252. Read data from device into CM buffer until (a) end of information is reached, (b) a short PRU with record level 16 or 17 has been read, (c) a zero-length logical record (any record level) has been read, or (d) the next PRU will not fit into the CM buffer. This is used by the CPU Loader when reading a relocatable binary file, since it does not stop at an ordinary end of logical record.

04 O.WRT — Corresponds to the WRITE system macro, CIO codes 014 and 016. Write data from CM to device until the CM buffer contains less than a full PRU.

05 O.WRTR — Corresponds to the WRITER system macro, CIO codes 024 and 026. Write data from CM until the CM buffer is empty, ending with a short (zero-length if necessary) PRU with level number specified in the request. If the "eof" flag bit is set, corresponds to the WRITEF system macro, CIO codes 034 and 036. Same as above, but the short PRU is followed by a zero-length level 17 logical record (a logical end of file mark).

9.5.2. Peripheral Processor Read/Write Orders

10 O.RDP — Same as O.READ, except read data from device into the requesting PP's memory.

11 O.RDPNP — Same as O.RCMPR, except read data from device into the requesting PP's memory. Used for loading mass storage resident PP programs and overlays.

14 O.WRP — Same as O,WRT, except write data from requesting PP's memory to device.

15 O.WRPR — Same as O.WRTR, except write data from requesting PP's memory to device.

9.5.3. Positioning Orders

12 O.SKF — Corresponds to the SKIPF system macro, CIO codes 240 and 242. Skip forward until N short PRUs with level $\geq$ the

level specified in the request have been read, or end of information is reached. With N = 777777B, the file is positioned at end of information.

13 O.SKB — Corresponds to the SKIPB system macro, CIO codes 640 and 642. Skip backward one or more PRU until N short PRUs with level $\geq$ the level specified in the request have been read, and then move forward over the last of these. With N = 777777B, the file is positioned at beginning of information (rewound).

16 O.BPRU — Corresponds to the BKSPRU system macro, CIO codes 044 and 046. Skip backward N PRUs. Note that this repositioning is by physical record units rather than logical records.

17 O.RCHN — Corresponds to the EVICT system macro, CIO codes 114 and 116. Release all record blocks assigned to the file (by clearing their RBR bits) and return the file's RBT word pairs to the RBT empty chain. If now possible, the size of the RBT area of CM is reduced. If the request references an FST entry, restore it to the state it would have after a REQUEST function; i.e., zeros in bytes 1 and 2 (RBT pointers), EST ordinal if a private unit or zero otherwise in byte 3, and allocation style code in byte 4.

9.6. Error Codes

This section describes the error conditions that can be detected by the Stack Processor and how it reacts to them. In some cases, the action depends on debug mode; i.e., the value of IP.DEBUG when 1SP was assembled. With IP.DEBUG = 0, these conditions are treated similarly to other errors: place error code in the code and status field of FET and FST entries and abort control point if error processing (EP) bit in FET is zero. With IP.DEBUG $\neq$ 0, issue an invalid MTR function with the 1SP output register having 77B in byte 0 and an error code in byte 1.

9.6.1. End of Information: This is not really an error condition. Put 01 into bits 9-13 of code/status but do not issue a message nor abort control point.

9.6.2. Parity Error: A parity error is reported whenever any possibly recoverable device error occurs during a read or write operation. These include actual parity error, lost data, mispositioning, and dropping ready during data transmission. The PRU is reread or rewritten up to a total of 62 attempts (3 for ECS). Whether success is attained or not, execution of the request continues after setting a flag. When execution of the request is completed, or the request is about to be reissued to the stack, the flag is examined. If the error was recovered, call 1SX with code 03 (dayfile message RECOVERED PARITY ERROR), but do not affect code/status nor abort the control point. If all 62 attempts had failed, call 1SX with code 04 (dayfile message UNCORRECTABLE PARITY ERROR), put 04 into bits 9-13 of code/status, and abort control point if EP bit is zero.

9.6.3. Invalid RBR Number: This occurs when processing a request that references an RBT which contains an RBR number greater than N.RBR - 1. In debug mode, issue bad MTR request (77B, 05B). Otherwise, call 1SX with code 05 (dayfile message NON-EXISTANT RBR REQUESTED), put 05 into bits 9-13 of code/status, and abort control point if EP bit is zero.

9.6.4. Disk Full: This occurs when processing a write request and a new record block is needed but none can be found. If EP bit is one, put 10B into bits 9-13 of code/status and terminate the request. If EP bit and control point error flag are both zero, call 1SX with code 76B (display WAITING — DISK FULL) and reissue request with bypass count set to 3. If EP bit is zero and control point error flag is non-zero, call 1SX with code 10B (dayfile message WAITING — DISK FULL) and terminate the request.

9.6.5. Buffer Parameter Error: This occurs when processing a request that references an FET and the following conditions are not all satisfied:

$$0 \leq \text{FIRST} < \text{LIMIT} \leq \text{field length}$$
$$\text{FIRST} \leq \text{IN} < \text{LIMIT}$$
$$\text{FIRST} \leq \text{OUT} < \text{LIMIT}.$$

Call 1SX with code 11B (dayfile message BUFFER PARAMETER ERROR), put 11B into bits 9-13 of code/status, and abort control point if EP bit is zero.

9.6.6. Not Assembled for ECS: This occurs when a request references a DST entry for allocatable ECS (device type AX), but driver overlay 3SX was assembled with IP.ECNOM = 0. 1SP issues a bad MTR request (77B, 00B).

9.6.7. Undefined Order Code: A request contains order code 06 or 07. Call 1SX with code 22B (dayfile message INVALID STACK ENTRY), put 22B into bits 9-13 of code/status, and abort control point if EP bit is zero.

9.6.8. Write Impossible: A write request does not reference an FST entry and has no RBT pointer and has zero in the "exact" flag bit (i.e., is not a rewrite in place operation). An RBT chain must be started, but there is no FST entry in which to store a pointer to the chain. In debug mode, 1SP issues a bad MTR function (77B, 22B). Otherwise, call 1SX with code 22B (dayfile message INVALID STACK ENTRY), put 22B into bits 9-13 of code/status, and abort control point if EP bit is zero.

9.6.9. Invalid RBT Address: An RBT pointer in a request, or in the FST entry referenced by the request, is out of range; i.e., greater than the number of word pairs in the RBT area of central memory. In debug mode, 1SP issues a bad MTR function (77B, 77B). Otherwise, call 1SX with code 22B (dayfile message INVALID STACK ENTRY), put 22B into bits 9-13 of code/status, and abort control point if EP bit is zero.

9.6.10. Connect Reject: A request references an I/O unit that cannot be connected or is not ready. If the control point error flag is zero, call 1SX with code 73B (display REJECT — ee STATUS xxxx yyyy) and reissue the request with bypass count set to 3. Otherwise, terminate the request with no message.

9.6.11. RBT Space Needed: This is not really an error condition. A new RBT word pair is needed but the RBT empty chain has no members. If the control point error flag is zero, call 1SX with code 77B (get more RBT space from MTR or display WAITING — RBT STORG), and reissue the request with bypass count set to 3. Otherwise, terminate the request with no message.

9.7 System Interface

This section enumerates the system tables, routines, MTR functions, and common decks used by the Stack Processor, and the residency requirements for 1SP and its overlays.

9.7.1. Tables

1. Control Point Areas: the control point error flag and storage move flag, and the CM reference address and field length. No interlock needed; the Stack Processor never changes these fields. CM RA and FL fields for control point zero are used as for other control points.

2. Device Status Table (DST): all fields of the DST entry whose ordinal is given by MTR in the 1SP input register. No interlock needed, since only MTR changes the first word and only the assigned 1SP changes the second word. In other entries, only the active flag (byte 4 of the second word) is examined.

3. Equipment Status Table (EST): the mass storage flag (bit 59), unloaded flag (bit 53), private flag (bit 52), off flag (bit 23), and DST ordinal (byte 4). No interlock needed; Stack Processor never changes these fields.

4. File Environment Table (FET): code and status field (bits 0-17 of first word), error processing flag (bit 44 of second word), and IN and OUT pointers (third and fourth words). No interlock needed, since the code and status is marked busy (even) before a request enters the stack, and is marked complete (odd) by Stack Processor only when execution of the request is finished.

5. File Status Table (FST): RBT/RB/PRU position pointers in bytes 1-4 of first word, and code and status field (bits 0-17) in second word. No interlock needed, for the same reason as with FET.

6. Record Block Reservation (RBR) area: all of the first header word (equipment type code, DST ordinal, unit number, starting RB number, and admissable allocation style codes), and bytes 1 (EST ordinal) and 3 (available RB count) in second header word; also individual RB bits in the remaining 36 words are toggled as record blocks are assigned and released. No interlock needed, because the only fields that can ever be changed (available RB count and individual RB bits) can be changed only by the Stack Processor assigned to the same DST ordinal.

7. Record Block Table (RBT): all fields. No interlock needed when operating on a file's RBT chain, for the same reason as with FET. The pseudo-channel CH.RBT is reserved only when RBT word pairs are being added to or removed from the RBT empty chain.

8. PP Resident words R.CPRA and R.CPFL: contain CM reference address / 100B and field length / 100B, respectively, for the control point to which the Stack Processor is currently attached.

9. PP Resident word R.STBMSK: contains appropriate mask when calling R.STB; always returned to its normal value (7700B).

10. CM word T.MSC: PP Job Queue entry count in byte 0, looked at but not changed by Stack Processor.

11. ECS Tables: ECST, ICEBUF, and LRD are used by the ECS driver overlay 3SX. Section 19. 8. 0 contains details. No interlocks are needed because only one copy of 1SP + 3SX can be active at a time.

9.7.2. Routines

1. PP Program 1SX: the Stack Processor Auxiliary program is called by 1SP (via M.RPJ) to take care of various exceptional conditions (described in section 9.6 above) that 1SP cannot handle itself or does not have time to do itself. For example, 1SP cannot issue dayfile messages because if the dayfile buffer is full, 1SP and MTR would loop endlessly waiting for each other. When more RBT space is needed, 1SP calls 1SX to get more RBT space from MTR, while 1SP proceeds to execute requests that do not require RBT expansion.

2. PP Program MTR: the System Monitor invokes 1SP initially (when a request entering the stack references an inactive DST entry), supervises the addition of all requests to the stack, and performs various functions for 1SP (see section 9.7.3 below).

3. PP Resident routine R.DCH: releases a channel reservation.

4. PP Resident routine R.IDLE: entered when 1SP releases its PP.

5. PP Resident routine R.MTR: used for all MTR functions other than reserve or drop channel.

6. PP Resident routine R.OVL: used for loading driver overlay 3Sx and non-standard allocation overlay 7Sx.

7. PP Resident routine R.RCH: reserves a channel.

8. PP Resident routine R.STB: used for inserting controller equipment number into device function codes and channel number into I/O instructions.

9. PP Resident routine R.TFL: used for computing an absolute CM address from one that is relative to a control point's RA, and to check whether a relative CM address is within the control point's FL.

10. CP Resident routine ICEBOX: used by the ECS driver overlay 3SX to transfer data between ECS and ICEBUF.

9.7.3. Monitor Functions

1. M.CCPA: used to switch 1SP from control point zero to the control point of a request, when necessary to interlock CM storage moves during execution of the request; also used when terminating any request (except those at control point zero) to switch 1SP back to control point zero and to decrement the request stack entry count component of the control point's activity count.

2. M.DPP: releases PP assignment.

3. M.EREQS: used to reissue a request to the stack. Only the special form, with 77B in byte 2, is used by 1SP.

4. M.ESTZ: used to mark a unit active when a file is assigned to the unit. Only the form with 10B in byte 2 is used by 1SP.

5. M.ICE: used by the ECS driver overlay 3SX to initiate central executive 2 (ICEBOX).

6. M.RBTSTO: used to reduce the size of the RBT area when all of the last 100B words of the area belong to the RBT empty chain.

7. M.RCH: used (rather than R.RCH) with zero in byte 4, to reserve the pseudo-channel CH.RBT only if it is immediately available.

8. M.RPJ: used for calling 1SX to another PP.

9. KILLMTR (77B): used for aborting the system when an impossible error condition is detected.

9.7.4. Common Decks

1. IPARAMS: the following installation parameters are used by 1SP:
IP.DEBUG — determines whether certain error conditions are to be treated like other errors or will cause 1SP to issue a bad MTR function.
IP.ECLRB — used by 3SX to calculate an absolute ECS address from logical RB and PRU numbers.
IP.ECNOM — if zero, causes only a dummy 3SX to be assembled; otherwise, determines the size of the LRD.

2. ECSCOM: the following BNL ECS parameter is used:
E.ALLOC — used by 3SX initialization to get allocatable block pointer word from ECST.

9.7.5. Residence Requirements

Program 1SP and overlay 7SZ must be CM resident. The 3Sx driver overlay(s) for the system residence device(s), and for all devices that may be used for system residence, must also be CM resident. Driver overlays for device types not included in the installation's configuration may be deleted from the system. Non-standard allocation overlays 7Sx must be CM resident if they are to be used; otherwise they may be deleted from the system.

9.8. PP Memory Allocation

The Stack Processor PP memory is divided into the following areas: direct cells, PP Resident, low-core working storage (overlaying initialization code), main program 1SP, middle-core working storage (also overlaying initialization code), driver overlay 3Sx, non-standard allocation overlay 7Sx, unused area, and high-core working storage (including I/O buffer area shared with overlay 7SZ). Figure 9-1 shows these areas and their approximate locations. Each area is described below.

9.8.1. Direct Cells (0000-0077): Cells 00 through 17 are used as local scratch cells by many subroutines, and cells 75, 76, and 77 contain pointers to the PP's communication area in CMR. Except for these, the direct cell arrangement used by Stack Processor is different from most other SCOPE 3 PP programs. Figure 9-2 shows direct cells 17-47, and cells 50-77 are

shown in Figure 9-3. The only infraction of SCOPE 3 convention is the use of D.PPIR (0075) to hold the constant 0500B; it is restored to its normal value (D.PPOR-1) before 1SP releases the PP.

9.8.2. PP Resident (0100-0775): PP Resident routines and tables used by Stack Processor are mentioned in section 9.7. When 1SP is loaded, its PP program header word is stored in cells 0773-0777; contrary to SCOPE 3 convention, the last two of these five bytes are used as working storage when 1SP is running.

9.8.3. Low-core Working Storage (0776-1056): When 1SP is first loaded, cells 0776 and 0777 contain the last two bytes of the PP program header word, and cells 1000-1056 contain part of the initialization routine. After initialization is completed, this area is used for working storage as described below.

1. NUNITS (0776): contains the unit number (0 to 7) of the largest-numbered unit on the controller described by the DST entry to which this 1SP is assigned.

2. NRBRS (0777): contains the value of N.RBR at the time CMR was assembled.

3. POOL (1000-1043): is a table containing NRM entries (rooms) of three bytes each. At present NRM = 12. Each room is either empty or contains some information about a request stack entry. This information includes a pointer to the full two-word request in the stack area in CM, and information for use in selecting the next request to be executed. Entries are added to POOL by the stack search routine and deleted by the request termination routine.

4. RBTWRD (1044-1056): is an eleven-byte area containing the first or current RBT word pair for the current request in the first ten bytes. The eleventh byte is a copy of the first byte (next word pair in chain, or zero).

5. NSRUN (1044-1053): is the first eight bytes of the RBTWRD area. During execution of the request selection procedure, byte NSRUN + i contains the number of requests in POOL for unit i, $0 \leq i \leq$ NUNITS.

9.8.4. 1SP Code (1057-approx. 6077): This is the non-initialization part of the main program, 1SP.

9.8.5. Middle-core Working Storage (approx. 6100–6121): This area overlays part of the 1SP initialization code, the 3Sx overlay header word, and the first two data bytes of overlay 3Sx. The first data byte contains the first byte address of the 3Sx linkage parameter list, and the second contains the first byte address of an R.STB list for insertion of the controller equipment number into device function codes. After the driver overlay 3Sx is loaded and initialization completed, this area is used for working storage as described below.

1. UNIT (approx. 6100–6107): is an eight-byte array giving the current position of each mass storage unit on the controller. Byte UNIT + i contains the current position, in standard record block designator format, of unit i, $0 \leq i \leq$ NUNITS. If the current position of a unit is not yet known, its byte contains zero.

2. SCRAP (approx. 6110–6121): is a ten-byte area which usually contains the current two-word request stack entry. It is used, rather than the copy in direct cells 52–63, when the CM copy of the request is to be modified without changing the direct cell copy.

9.8.6. 3Sx Code (approx. 6122–6720): This is the non-initialization part of the driver overlay 3Sx. For each driver, the symbol .NSAORG is the last byte address plus one of this area.

9.8.7. 7Sx Code (approx. 6721–6744): This area contains the overlay header word and code for whatever non-standard allocation overlay, if any, is in core at the moment. The total size of a 7Sx overlay, including the overlay header word and initialization code, must not exceed the smallest value of .PATCH for all of the driver overlays with which the NSA overlay will be used.

9.8.8. Unused Area (approx. 6745–7151): This area is available as patch space. At present, the smallest unused area size (approx. 205B bytes) is with 1SP + 3SQ + 7SB; i.e., the 6638 disk driver with RESPOND interface.

9.8.9. High-core Working Storage (7152–7777): The 3Sx initialization code may extend into this area but after initialization is completed, bytes 7152 (DRIVEND) and above are used for working storage as described below.

1. SXDATA (7152–7170): is a fifteen-byte (three CM words) area containing information that is written to the 1SP message buffer for use by program 1SX when a device error is detected. A detailed description is given in the 1SX section of this document.

2. RBCOUNT (7171): contains the maximum number of record blocks to be processed by the current request before it is reissued to the stack. It is set by .SHIFT and used by .POSIT in those driver overlays in which the symbol MAXRBCT was SET by the installation to some value less than the number of record blocks per cylinder for that device type.

3. ORDLEV (7172): contains the record level specified in bits 18–21 of the first word of the current request. It is significant only during execution of order codes O.RDSK, O.WRTR, O.WRPR, O.SKF, and O.SKB.

4. RBREST (7173): contains the EST ordinal from byte 1 of the second header word of the RBR currently referenced. It is set by RBRDEV in 1SP and used by PRURDZ and PRUWTZ in 7SZ.

5. IOBUF (7174–7677): contains the PRU currently being transmitted to or from the mass storage unit. The first byte is not used. The second byte (WCTA) contains the length of the PRU in bytes. The next 500B bytes (WORD0 and following) contain the 320 bytes = 64 CM words of data in the PRU. For a short PRU byte WORD0+(WCTA) contains the logical record level in binary form, and the remaining bytes are unused. The last two bytes of the IOBUF area are used only with 6603-II and 6638 disk units; they contain the physical address of the PRU and are used to verify unit positioning. When the IOBUF area does not contain a PRU of data, it may contain overlay 7SZ. In this case, the overlay header word is loaded starting at IOBUF.

6. HOLDER (7700–7777): is used only during execution of an O.SKB request. It contains the PRU numbers of all short PRUs in the current RB having record level ≥ the level specified in the request. Since the longest possible RB is 64 PRUs (6603 disk outer zone), HOLDER is 64 bytes in length.

9.9. Narratives

This section describes the individual routines that comprise the Stack Processor package. The routines in overlay 7SZ are extensions of 1SP routines and therefor are not described in a separate section. The routines are grouped into major sections of code, which are described below in the order in which they appear in the assembly listing.

Initialization: Executed once when 1SP is loaded. Sets up all constants, pointers, etc. that will not change until 1SP releases its PP. Loads the 3Sx driver overlay. All of the initialization code is overlaid by working storage and 3Sx code.

Search of Request Stack: Executed after Initialization and each time execution of a request is terminated. Searches the request stack and constructs a POOL entry for each request that is a candidate for execution. If an immediate request (i.e., one that requires no physical device access) is found, it is executed as soon as it is found.

Selection of Best Request: Executed after Stack Search if no immediate request was found. Examines all POOL entries and chooses one request for execution, based on request priority and device access time comparisons.

Initiation of Request Execution: Executed when Stack Search finds an immediate request or when Selection has chosen the optimal non-immediate request. Performs initialization of working storage and modifies instructions where necessary, to prepare for execution of the request, and then jumps to the executive routine specified by the order code.

Request Executives: A group of routines, one for each order code, that perform the actual execution of requests. They call read/write subroutines, manipulate buffer and file pointers, etc. as needed for each order code.

Termination of Request Execution: Entered when an executive determines that execution of the current request is either completed or must be suspended temporarily (by reissuing the request to the stack). File and buffer pointers

and code/status fields are updated. If necessary, the PP is released; otherwise, control goes back to the Stack Search section.

Advance Record Block Subroutines: Called by the executives when a record block boundary is reached, to find or allocate the next record block. The three subroutines are for backspacing, reading, and writing.

Miscellaneous Subroutines: This section contains closed subroutines that do not belong in any one of the above sections.

Driver Overlays: Contain all code that varies among device types. Like-named routines in each driver perform similar functions, and appear in the same order in all driver overlays.

9.9.1. Initialization

The 1SP initialization procedure is in two parts. When 1SP is loaded, R.IDLE jumps to C.PPFWA (1000) which contains a jump to the first part of Initialization beginning at OVER (approx. 6100). This code does all initialization that can be done before the driver overlay 3Sx is loaded. It then jumps to the second part of Initialization beginning at INITIAL (1002) which loads 3Sx and does initialization that must be done after 3Sx is loaded.

OVER: setup constant TWO, read P.RBR and P.RQS into RBRPWD and STLWD respectively, and store the EST base address from P.EST into LOCEST.

OVERB: compute number of RBRs = (T.DST − T.RBR) / RBRSIZE and store into NRBRS.

OVERC: adjust the STASA and STACT bytes of STLWD because the rest of the system uses zero-origin indexing for request stack entries but 1SP uses one-origin indexing so that an empty POOL room contains zero. Read the assigned DST entry into DSEWD1 and DSEWD2 (the assigned DST ordinal is found in cell 22 because R.IDLE reads the input register into 16−22 so the program name is where R.OVL expects it).

OVERE: search the EST and store into NUNITS the unit number of the largest-numbered mass storage unit on the assigned controller. This is used to eliminate unnecessary passes through the outer loop in the Selection section.

OVERA: move linkage parameter list PARAMS to high core so it will not be destroyed when3Sx is loaded. Save EQUIP byte of DSEWD2 so this cell can later be used with CSTSWD. Store HPOS1 and HPOS2 into the first two bytes of UNIT and zeros into the remaining six bytes. Setup constant MAXPRU. Store driver overlay name 3Sx into cells 16-17 for R.OVL.

OVERF: if the device has very tight timing (currently, the 6603-II and 6638 disks qualify), modify some instructions so that the time lost to PP pyramid conflicts between PRUs is minimized if the Central Memory Access Priority (CMAP) optional feature is installed.

OVERD: put 3Sx load address into A-register for R.OVL and go to INITIAL.

INITIAL: call R.OVL to load driver overlay 3Sx. Call MODIFY to process the linkage parameter lists; the primary purpose is to store locations defined in 3Sx into their points of reference in 1SP. Call R.STB to insert controller equipment number into bits 9-11 of all device function codes. Set flag saying no non-standard allocation (NSA) overlay 7Sx is in core.

SETCHAN: this code is stored from the linkage parameter list because it depends on device type. For 6000-series devices, call R.STB to store the channel number into all I/O instructions, since only one channel can be used. For 3000-series devices, store the channel number(s) into .CHRQA for later use since the device may be attached to two channels and, if so, it is fastest to use whichever channel is available when one is needed. For ECS, store the channel number(s) for the same reason, and go to XINIT (in 3SX) to setup the absolute base address of the allocatable portion of ECS; return to INITA.

INITA: store zero into CUR, so that HPOS1 = UNIT + (CUR).

INITLP: this code is located in what will later be the RBTWRD working storage area. Zero out the code just executed, which now becomes the POOL working storage area, and go to ORDEXE in the Termination section, to prepare for Stack Search.

9.9.2. Search of Request Stack

This procedure searches the request stack area of CM to find all requests that are candidates for execution by this 1SP. Each such request is recorded

in POOL if not already there. When an immediate request (one involving no physical device access) is found, stack searching is discontinued and execution of the request is initiated. Otherwise stack searching ends when all relevant requests have been found or POOL is full, and control passes to the Selection section. The code from VERI to VERIX is executed twice for a non-immediate request: once during stack searching and again when the request has been selected for execution.

STAKAA: upon entry CTA contains the number of request stack entries that reference this 1SP's DST entry. The stack is searched from high core to low, so CXAD is set to point to the high end.

POOLA: find next vacant POOL room and store its room number in RMNR.

POOLB: if POOL is full, go to Selection.

POOLD: if CTA is zero, go to Selection.

POOLE: decrement CXAD and look at requests until one is found containing the DST ordinal for this 1SP, then decrement CTA. Go to POOLD (i.e., ignore the request) if it is already in POOL (bit 11 of first word of request is set). If the bypass count (bits 9-10 of first word) is non-zero, decrement it and go to POOLD to ignore the request until next stack search time.

POOLG: the request is a candidate for execution by this 1SP and is not in POOL yet. Store CXAD (pointer to request in stack) into POOL entry.

VERI: store control point number from request into OCTLPT. Store zero into ORDTYPE if the request is a write but not rewrite in place operation, otherwise store non-zero. Thus, ORDTYPE = 0 means the request is capable of allocating record blocks and RBT word pairs. Setup default values for last RB length, allocation style code, and code/status. Go to STKCPLA if the request contains an FST address. If RBT pointer is also absent, report end of information (9.6.1) or write impossible (9.6.8) depending on ORDTYPE; otherwise go to STKCPLG.

STKCPLA: save the FST address in FSTADR, and read the two FST words to get the RBT pointers and code/status. If the first and current RBT pointers are both zero, go to STKCPLB. If one of them is zero, make it equal to the other. Go to STKCPLH if first RBT pointer is out of range. Read the first RBT word pair and go to STKCPLF.

STKCPLB: FST present but no RBT chain. Report end of information or go to STKCPLZ (in 7SZ) depending on ORDTYPE.

STKCPLZ: call LOCATE to find a suitable RBR for the file. If none was found, report disk full (9.6.4); otherwise go to STKCPLD.

STKCPLD: reissue the request if the located RBR has another DST ordinal. Otherwise, call NRBTW to get an RBT word pair from the empty chain or reissue the request if none is available. Setup the file's first RBT word pair and make the FST point to it.

STKCPLF: enter with first RBT word pair in RBTWORD area. Store allocation style code and last RB length into ALLOC and EOIPRU respectively.

STKCPLG: all paths rejoin here. Check current RBT pointer for validity; go to STKCPLX if in range.

STKCPLH: report RBT address out of range (9.6.9).

STKCPLX: read current RBT word pair into RBTWRD area. Find current RB byte if ORDTYPE is non-zero, otherwise skip to end of information

VERIAC: call ADRCOM to get the current RBR number and RB designator. If the latter is zero, call SEEKRB and GETPOS to find and standardize an available RB. The RB is not actually allocated; i.e., its RBR bit is still 0.

VERIAB: store RB designator (standardized if necessary), unit number, and priority flag bit into POOL entry. Reissue the request if the current RBR references another DST entry.

VERIC: if VERI was entered from the Selection section, or if the request is immediate, go to VERIX; otherwise, go to VERIE. A request is immediate if it is one of the following:

O.SKF with N = 777777B (skip to EOI),
O.SKB with N = 777777B (rewind file),
O.BPRU (backspace PRUs), or
O.RCHN (release chain).

VERIX: go to Initiation to begin execution of the request.

VERIE: set bit 11 of the first word of the request in CM to show that it is in POOL, call .SECADR to get the starting physical sector number and store it in the POOL entry, zero the request subpriority, and go to POOLB to continue stack searching.

9.9.3. Selection of Best Request

This section is entered when Stack Search finishes without having found an immediate request. Selection chooses one request from among those in POOL, using device access time and request priority as criteria, and exits to VERI in the Stack Search section, to prepare for execution of the chosen request.

SELECT: search POOL and store the number of requests for unit i into NSRUN+i, $0 \le i \le$ NUNITS. It is possible that POOL is completely empty if all request stack entries were discarded or bypassed for various reasons. If so, (CUR) is still 7776B from the loop that cleared NSRUN; go to ORDEXE to prepare for Stack Search again. If POOL is not empty, call .ACTIVC to reserve a channel and store the current angular positions into ANGPOS. Exception: for ECS, a channel is not reserved until initiation of request execution, and then only for a PP I/O request.

SELECTD through SELECTM is the outer loop, executed once for each unit 0 to NUNITS; CUR contains the current unit number. If NSRUN+(CUR) is zero the loop does nothing. Otherwise, store head position from UNIT+(CUR) into HPOS1 for easy access by 3Sx.

SELECTF through SELECTJ is the inner loop, executed once for each POOL entry; RMNR contains the current room number. If the room is vacant or contains a unit number other than (CUR) the loop does nothing. Otherwise:

Store the request priority and subpriority in a scratch cell. Subpriorities 0-6 are stored as 0; thus there are four priority groups:

17	priority 1, subpriority 7
10	priority 1, subpriority 0-6
07	priority 0, subpriority 7
00	priority 0, subpriority 0-6.

The request is ignored if its priority group is less than that of the "best" request found thus far for unit (CUR).

SELECTH: call .HEAD to compute the access time (latency only for drums, head motion and latency for 6638 disks, head motion only for other disks, and zero for ECS). If this request's priority group is greater than that of the "best" request thus far for unit (CUR), this request is now the new

"best" request thus far for unit (CUR), regardless of access time. If the priority groups are equal, the request with the smaller access time becomes the new "best" request thus far for unit (CUR).

SELECTK: inner loop completed; the "best" request for unit (CUR) has been found. Store the new head position into UNIT+(CUR) and call .SHIFT to initiate head motion if necessary. The outer loop chooses the "best" request across all units, by a method similar to that used in the inner loop to choose the "best" request for one unit.

SELECTN: outer loop completed; the request to be executed now has been chosen. For each POOL entry, add 1 to the subpriority if it is not already 7.

SELECTR: extract unit number and stack location from chosen POOL entry and store into CUR and CXAD respectively. Store head position from UNIT+(CUR) into HPOS1 for easy access by 3Sx. Read the full two-word request stack entry. Clear bit 11 of the first word of the CM copy of the request to show it is no longer in POOL. Exit to VERI (in Stack Search section) to get ready for request execution.

Note that each time Selection is executed, .SHIFT is called exactly once for each unit for which POOL contains at least one request. This ensures maximum use of the "seek overlap" capability of a subsystem such as a 3234 controller with up to eight 854 disk pack units.

9.9.4. Initiation of Request Execution

A request has been chosen for execution. This section is entered from VERIX in the Stack Search section, does all initialization prior to actual execution of the request, and exits to the appropriate executive routine.

EXEC: call ASSIGN to attach 1SP to the control point of the request if appropriate. For ECS only, go to XGETCH (in 3SX) to reserve a channel if the order code specifies PP I/O, and return to STAKCB.

STAKCB: call MODIFY to setup various instructions and switches for reading or writing depending on the order code. Call .RDINST or .WTINST to setup the driver overlay for reading or writing. Go to RWCMOD if the order code is non-write.

WTINIT: order code is some kind of write. If the "exact" flag bit is zero, go to WTINSTA. Otherwise, the request is a rewrite in place operation; Stack Processor becomes slightly schizophrenic in this case, with the .POSIT and SECOMP routines setup for reading and all other code for writing.

WTINSTA: call SECOMP to determine length of current record block.

RWCMOD: all paths rejoin here. For a CM I/O or positioning request, go to RWCMODB. For a PP I/O request, store the channel number (to be used for transmitting data between 1SP and the requesting PP) into the communication word, and set the control flag to 1. Upon sensing this, the requesting PP will store the channel number into its own IAM/OAM instruction and acknowledge by setting the control flag to 2. Initialization for PP I/O is completed; go to GOTOIT to jump to the proper executive.

RWCMODB: order code is not PP I/O. Zero some work areas. For a positioning request, initialization is completed; go to GOTOIT to jump to the proper executive. The remainder of this section, down to GOTOIT, performs initialization for a CM I/O request. Test for FIRST $<$ LIMIT and LIMIT $\leq$ field length and report buffer parameter error (9.6.5) if either test fails. For CM I/O with some devices, inter-PRU timing is very tight. The remaining code presets many instructions in the CEMIO and RDMSTR subroutines (see 9.9.5.1) to minimize their execution time.

STAKCC: go to RWCMODC if an FET is present. Otherwise, set switches in CEMIO for the no-FET case, set AFLD (IN for reading, OUT for writing) equal to the first word address of the CM buffer, and go to RWCMODE.

RWCMODC: setup LDC instructions in CEMIO with the absolute addresses of the IN and OUT pointer words, and read the appropriate word (IN for reading, OUT for writing) into AFLD.

RWCMODE through RWCMODN: setup various instructions in CEMIO with the absolute FIRST and LIMIT addresses for the CM buffer area.

RWCMODF: test for FIRST $\leq$ AFLD and AFLD $<$ LIMIT and report buffer parameter error (9.6.5) if either test fails.

RWCMODI: setup switches in the CEMIO and RDMSTR subroutines depending on the "no-FET" and "exact" flag bits.

RWCMODJ: setup switches in RDMSTR if the order code is O.RDNS, because ordinary short PRUs do not stop execution of this order.

GOTOIT: all paths rejoin here. Using the order code as index, get address of proper executive routine from the table ORDRAD and jump to that routine in the Executives section.

9.9.5. Request Executives

This section performs the actual execution of requests. The routines are arranged in three groups: CM read/write, PP read/write, and positioning. Each routine exits to the Termination section.

9.9.5.1. CM Read/Write Executives.

These routines control all transmission of data between mass storage and central memory. Their common code is in two subroutines, RDMSTR and WRMSTR, which in turn call subroutine CEMIO and subroutines in 3Sx.

RCMPRO: executive for O.RCMPR orders. Set switches in CEMIO to skip the first three CM words of the first PRU read, change order code to O.READ, and fall through to READ.

READ: executive for O.READ and O.RDNS orders. Set switch in RDMSTR to backspace upon reading a PRU too large for the CM buffer, and call RDMSTR. If execution was completed, go to Termination. If stopped by buffer full and order is O.RDNS and last PRU transmitted to CM was short, store zeros in WDCT and WORD0 so that the code/status will show that the file is positioned between logical records, and then go to Termination.

RSKP: executive for O.RDSK orders. Set switch in RDMSTR to exit without backspacing upon reading a PRU too large for the CM buffer, and call RDMSTR. Go to RSKPB if a short PRU was read and all of it was transmitted to CM. Otherwise, call CEMIO to transmit as much of the PRU as will fit into the CM buffer, then go to RSKPC if it was not a short PRU.

RSKPB: short PRU found. If its level is $\geq$ the level specified in the request, go to Termination.

RSKPC: store 1 into LEVCT, change order code to O.SKF, and go to that executive routine (section 9.9.5.3, label SKPFB).

WRIT: executive for O.WRT orders. Call WRMSTR and go to Termination.

WRTR: executive for O.WRTR orders. Call WRMSTR to write as many full PRUs as are in the CM buffer, then call CEMIO and .RDISK to get and write a short PRU whose length is the remaining buffer content (possibly zero) and whose level is that specified in the request, and go to Termination.

Subroutine RDMSTR: common code for all reading from mass storage to central memory. Call SECOMP to determine length of current record block. To save execution time in inter-PRU processing, setup return address for .POSIT, .RDISK, and CEMIO so each jumps directly to the next; this eliminates three RJM instructions.

RDMSTRA: go to .POSIT to setup for next PRU, thence to .RDISK to read the PRU into IOBUF, and then to CEMIO to send the PRU to the CM buffer if it will fit. If it did fit, go to RDMSTRC. If order code is not O.RDSK, backspace over the PRU. Return to caller with A-register negative to indicate reading stopped by buffer full.

RDMSTRC: if order code is O.RDNS and the PRU just sent to CM was the first of a logical record and had zero length, go to RDMSTRJ. Otherwise, if the PRU was short, go to RDMSTRH if order code is O.RDNS or to RDMSTRG for other orders. If the PRU was not short, go to RDMSTRE if the "no-FET" and "exact" flag bits are both set; otherwise cancel the first test at RDMSTRC and loop to RDMSTRA for the next PRU.

RDMSTRG: non-O.RDNS request execution completed by reading a short PRU; return to caller with A-register positive.

RDMSTRH: order is O.RDNS and a short PRU has been read. If its level is 16B or 17B, go to RDMSTRJ. Otherwise, reactivate the first test at RDMSTRC (because the next PRU will be the first of a logical record) and go to RDMSTRA for the next PRU.

RDMSTRJ: order is O.RDNS and a logical record with zero length and/or level 16B or 17B has been read. Store 17B as record level read (so code/

status will say end of file) and return to caller with A-register positive to indicate reading was stopped by a condition other than CM buffer full.

RDMSTRE: the "exact" and "no-FET" flag bits are both set, and a PRU is about to be read. If the CM buffer is completely full, return to caller with A-register negative; otherwise go to RDMSTRA for the next PRU. This kludge is required by the OPTIMA system, whose files do not end with short PRUs and sometimes have inaccurate end of information pointers.

Subroutine WRMSTR: common code for writing full PRUs from central memory to mass storage. To save execution time in inter-PRU processing, setup return addresses for .RDISK, .POSIT, and CEMIO so each jumps directly to the next; this eliminates three RJM instructions. Go to .POSIT to setup for next PRU and thence to CEMIO to get the PRU from CM, returning to WRMSTRC.

WRMSTRC: if CEMIO found that the CM buffer contains less than a full PRU, return to caller. Otherwise, go to .RDISK to write the PRU to the device, thence to .POSIT to setup for the next PRU, and then to CEMIO to get the next PRU from CM, returning again to WRMSTRC.

Subroutine CEMIO: handles all data transfer between IOBUF and a buffer in CM, one PRU at a time, and updates the IN or OUT pointer if an FET is present. Upon entry, WDCT contains the PRU length in bytes.

CEMIOC: if order is O.RCMPR and this is first PRU, go to CEMIOA to adjust instructions and WDCT to bypass the first 15 bytes of the PRU, and return to CEMIOT. Otherwise, compute the PRU length in CM words. If it is 500B bytes, this is 100B CM words (i.e., a full PRU); go to CEMIOD.

CEMIOT: the PRU is short; divide WDCT by 5.

CEMIOD: store PRU length in CM words into CMWDCT. If order is O.RCMPR and this is first PRU, go to CEMIOB to restore WDCT to its true value (for later short PRU tests) and return. If no FET is present, store LIMIT – 1 into BFLD. Otherwise, read the appropriate pointer word (OUT if reading, IN if writing) into BFLD. (This is done each time through CEMIO, so that Stack Processor and the requestor can chase each other around the circular buffer, processing data at device speed through many bufferfuls.) Test for

FIRST $\leq$ BFLD and BFLD < LIMIT and report buffer parameter error (9.6.5) if either test fails.

CEMIOG: compute the buffer capacity (reading) or content (writing) in CM words. This is the directed distance (in a circular sense) from (AFLD) to (BFLD), minus one if reading (since IN = OUT means buffer empty, not full).

CEMIOJ: this amount is less than (CMWDCT), so multiply it by 5 (to get byte count) and store result in CTA, and return to caller with A-register negative to indicate nothing was transmitted.

CEMIOK through CEMION: buffer wraparound may occur within this PRU because (AFLD) > (BFLD). Area 1 is the part of the PRU from (AFLD) to LIMIT and its length (in CM words) is stored in CTA. Area 2 is the part of the PRU starting at FIRST, and its length (in CM words) is stored in CMWDCT. Either or both may be zero, but their sum equals the PRU length in CM words.

CMIOMOD1: this is the CRM or CWM instruction that transmits the first part of the PRU to or from area 1 of the CM buffer. If area 2 does not exist, add the PRU size to (AFLD) and go to CEMIOS.

CMIOMOD2: this is the CRM or CWM instruction that transmits the second part of the PRU to or from area 2 of the CM buffer. Store FIRST + (CMWDCT) into AFLD.

CEMIOS: if no FET is present, go to CEMIOY. Otherwise, write the updated AFLD to the appropriate pointer word (IN if reading, OUT if writing) and go to CEMIOX.

CEMIOY: store the updated AFLD into the START field of the request stack entry, so that I/O will not start over from the beginning of the buffer if the request is reissued before it is completed.

CEMIOX: return to caller with A-register positive to indicate that a PRU was transmitted.

9.9.5.2. PP Read/Write Executives

These routines control all transmission of data between mass storage and the peripheral processor memories. Their common code is in subroutine PPIO.

When a PP issues a PP I/O request (order code 10, 11, 14, or 15), the request stack entry is in the first two words of the requesting PP's message buffer, and the third word is the "communication word" which is to PP I/O what the FET is to CM I/O. Figure 9-4 shows the formats of this word during processing of a PP I/O request. When the request enters the stack, the control flag is zero. When 1SP selects the request for execution, it sets the channel number and changes the control flag to 1. (The same data channel is used for transmission between 1SP and the mass storage device, and between 1SP and the requesting PP.) The requestor stores the channel number into its own IAM/OAM instruction and changes the control flag to 2. For reading: 1SP reads a PRU from the device, stores PRU length (in bytes) in communication word and changes control flag to 3, and sends (via OAM) the PRU to requestor, who then changes the control flag back to 2. For writing: 1SP stores PRU length (in bytes) in communication word and changes control flag to 3, accepts (via IAM) the PRU from requestor who then changes control flag back to 2, and 1SP then writes the PRU to the device. In either case, when the control flag is 2 and 1SP determines that execution of the request is completed, it stores a CIO-style code/status (including error code and record level) into the communication word and changes the control flag to 4. The requesting PP acknowledges receipt of the code/status by changing control flag to 5, after which 1SP changes the control flag back to 0 and processing is finished.

PPRD: executive for O.RDP and O.RDPNP orders. Call SECOMP to determine length of current record block.

PPRDA: call .POSIT and .RDISK to prepare for and read next PRU. If order code is even (i.e., not O.RDPNP) go to PPRDD. Adjust WDCT and the OAM instruction to skip the first three CM words (15 bytes) of the PRU, call PPIO to send the PRU to requesting PP, restore the OAM instruction, and go to PPRDC.

PPRDD: call PPIO to send the PRU to requesting PP.

PPRDC: upon return from PPIO, the A-register indicates what to do next. If A = 0, a full PRU was transmitted; go back to PPRDA for the next PRU. If A > 0, a short PRU was transmitted; go to Termination. If A < 0, the PRU

would not fit into the requesting PP's buffer and therefor was not sent; backspace over it and go to Termination.

PPWT: executive for O.WRP and O.WRPR orders.

PPWTA: call .POSIT to setup for next PRU. Call PPIO to get a PRU of data from the requesting PP. Upon return, non-zero in the A-register means the requesting PP has less than a full PRU so nothing was sent; go to PPWTB. Zero in the A-register means a full PRU was transmitted; call .RDISK to write it to device and go to PPWTA for next PRU.

PPWTB: if order code is even (i.e., not O.WRPR) go to Termination. Store remaining byte count of requesting PP's buffer (i.e., length of short PRU) into WDCT, store record level from request after last data byte, call PPIO to get short PRU from requesting PP and .RDISK to write it out, and go to Termination.

Subroutine PPIO: transmits data between Stack Processor PP and the requesting PP. Wait until requestor sets control flag to 2 to indicate that it is ready for data transmission. Read second word of request stack entry to get updated remaining byte count of requestor's buffer. If this is less than (WDCT), return to caller with A-register < 0 to indicate no data was transmitted. Write second word of request stack entry with remaining byte count reduced by (WDCT). Store (WDCT) into communication word for use by requesting PP with its IAM/OAM instruction. If (WDCT) = 0 go to PPIOC because there is nothing to transmit. Otherwise, change control flag to 3, so requestor will execute its IAM/OAM instruction. Activate channel, execute Stack Processor's IAM/OAM, and deactivate channel.

PPIOC: return to caller with A-register > 0 if a short (or zero length) PRU was transmitted, or = 0 if a full PRU was transmitted.

9.9.5.3. Positioning Executives

These routines perform execution of all positioning requests (order code 12, 13, 16, or 17).

SKPF: executive for O.SKF orders. If record count specified in request is 777777B, call SKPEOI to skip to end of information (by scanning the RBT

chain rather than actually reading all of the file) and go to Termination to report end of information (9.6.1).

SKPFA: not special case. Call SECOMP to determine length of current record block.

SKPFB: call .POSIT and .RDISK to setup for and read the next PRU. If it is not short, or is short but has record level < the level specified in the request, go back to SKPFB for the next PRU. Otherwise, decrement record count in the request. Go to Termination if result is zero; otherwise go back to SKPFB.

SKPB: executive for O.SKB orders. If record count specified in request is 777777B, rewind the file by resetting the current position pointers (RBT word pair, RB byte, and PRU numbers) and go to Termination to report beginning of information status.

SKPBC: not special case. If last record block length (EOIPRU) is zero, change it to 7776B. Call SECOMP to determine length of current record block. Backspace one PRU unconditionally. If current position was between record blocks (PRU was 0), call PRUBK with A-register negative to change position to last – 1 PRU of previous RB and go to SKPBAA. If current position after decrementing is between record blocks (PRU was 1), go to SKPBA.

SKPBAB: PRU was 2 or more. If it was less than current RB length, leave it as decremented. If PRU was 7776B (indicating initial backspace done), change it to RB length. If PRU was 7775B (indicating initial backspace not done yet), change it to RB length minus one. In any case, go to SKPBAA.

SKPBA: call PRUBK with A-register positive to set current position to last PRU of previous RB.

SKPBAA: A-register contains number of PRUs remaining, after the initial unconditional backspace of one PRU, from beginning of record block to current position. Store this in AFLD and zeros in BFLD and PRU.

SKPBB: call .POSIT and .RDISK to read PRUs 0 through (AFLD)–1 of the current RB. For each short PRU whose record level ≥ the level specified in the request, store the PRU number+1 in HOLDER+(BFLD) and incre-

ment BFLD. After completion of this loop, compare (BFLD) with the record count in the request. If less than, subtract (BFLD) from the record count and go to SKPBA to continue backspacing into the previous record block.

SKPBE: subtract record count from (BFLD), store the PRU number + 1 from HOLDER+(BFLD) into PRU, and go to Termination.

BPRU: executive for O.BPRU orders. If last record block length (EOIPRU) is zero, change it to 7776B. Call SECOMP to determine length of current record block.

BPRUA: decrement PRU. Go to BPRUCA if result ≥ 0, otherwise call PRUBK to set current position to last PRU of previous record block.

BPRUB: decrement PRU count in request and go to Termination if result is zero; otherwise loop to BPRUA.

BPRUCA: if (PRU) was greater than current RB length, set current position to last PRU of current RB. Go to BPRUB.

RCHN: executive for O.RCHN orders. Load overlay 7SZ if not already in core, and go to RCHNZ. This code is in 7SZ because it uses the record block releasing code (PRURDZ) which is also in 7SZ.

RCHNZ: set "release" flag bit in request if not already set, and call PRURD repetitively to scan and release the RBT chain. When end of information is reached, PRURD exits to Termination.

9.9.6. Termination of Request Execution

This section is entered to perform final bookkeeping when execution of a request is completed, or to reissue a request to the stack when its execution must be suspended for some reason. There are several entry points to the Termination section; the one used depends on the type of request being executed and the reason for completion or suspension of execution. Termination exits to Stack Search or releases the PP.

INVAL: entry point for invalid request stack entry (9.6.7, 9.6.8, 9.6.9). Load 1SX code 22B and go to EREXIT.

BUFPE: entry point for buffer parameter error (9.6.5). Load 1SX code 11B and go to EREXIT.

EREXITZ: entry point for end of information (9.6.1) reached during read, skip forward, or rewrite in place operation. Setup WDCT and WORD0 so code/status will indicate end of file, load code 01 and go to EREXIT.

EREXITY: beginning of information reached during backward positioning operation. Set WDCT to full PRU size (so bits 3-4 of code/status will not be changed), load code 01 and go to EREXIT.

EREXIT: general entry point for all errors that force completion of request execution. The error code is in the A-register. If entered from Stack Search, call ASSIGN to attach 1SP to the request's control point if needed. Store error code into bits 9-13 of code/status. If code is 01 (end or beginning of information), go to EREXITA. Setup abort flag (1 = abort, 0 = don't abort) for 1SX.

EREXITB: call CALL to call 1SX to another PP.

EREXITA: if the request being terminated is not a write, go to WRTEXX.

WRTEX: entry point for termination of a write request. If the last PRU written was short and the "eof" flag bit is set in the request, call .POSIT and .RDISK to setup for and write an end of file PRU (length zero, level 17B).

WRTEXX: go to ORDEX.

ERRCALL: entry point for conditions that require calling 1SX and reissuing the request with bypass count set to 3. These conditions are disk full (9.6.4) and connect reject (9.6.10).

BYPASS: entry point for conditions that require reissuing the request with bypass count set to 3 without calling 1SX. This causes the request to be ignored the next three times it is found by Stack Search. This entry point is used by subroutine ASSIGN when a control point's storage move flag is set, and by NRBTW when it cannot immediately find a free RBT word pair.

NEWDEV: entry point used when the request must be reissued to the stack with the DST ordinal changed (i.e., passed to another Stack Processor), without calling 1SX or setting the bypass count. This entry point is used by the Advance Record Block subroutines.

NOTDONE: general entry point for reissuing a request to the stack before its execution is completed. Clear the completion bit in code/status, so that later parts of Termination know that the request is not complete. If order code is O.SKB and (PRU) = 0, change it to 7776B.

ORDEX: all Termination entry paths rejoin here. Normally, location ORDEX contains a jump to ORDEXA. If a parity error (9.6.2) is detected during execution of a request, ORDEX is changed to a do-nothing instruction: 0003 if a retry attempt was successful or 0000 if all attempts failed. Restore ORDEX to its normal state. If ORDEX was 0003, go to EREXITB to call 1SX with error code 03 and abort flag 0. If ORDEX was 0000, go to EREXIT with error code 04.

ORDEXA: if the last PRU transferred was not short, go to ORDEXB. If the record level in WORD0+(WDCT) is 17B (end of file), store 3 into bits 3-4 of code/status, otherwise store 2 (end of record). In either case, store the record level into bits 14-17 of code/status.

ORDEXB: if the request contains no FST address, go to ORDEXBA. Store updated code/status into second FST word. If request is a write but not rewrite in place , and if last RB length is to be updated (701B means don't update it), store (PRU) as new last RB length into the first word pair in the file's RBT chain. Rewrite both FST words into CM with updated file position and code/status fields.

ORDEXBA: if the request references a reply word or FET, and if code/status completion bit is set, store updated code/status into bits 0-17 of the reply word or first FET word in CM.

ORDEXC: if request is not PP I/O, go to ORDEXCB.

ORDEXCW: termination for PP I/O requests. Read communication word and loop until control flag is even (i.e., 2). Store lwa+1 of data transmitted from communication word into buffer fwa in request stack entry, in case request is being reissued. Put updated code/status into bits 0-17 of communication word. If completion bit is set, change control flag to 4, otherwise make it 0. Write the updated request stack entry and communication word to requesting PP's message buffer. If request is being reissued, go to ORDEXCB.

ORDEXXX: read communication word and loop until requesting PP has acknowledged completion by changing control flag to 5, then rewrite the communication word with control flag zeroed.

ORDEXCB: all paths rejoin here. If request execution is completed, store zero into bits 0-11 of first word of request, to mark this word pair in stack as available. Whether complete or not, write both words to stack. If execution is completed, go to ORDEXDD.

REISSUE: issue MTR function M.EREQS (34B, x, 77B, sr, ds) where sr is the zero-origin index (from CXAD, location within stack area) for the request and ds is the DST ordinal to be stored in bits 0-5 of first word of the request. MTR stores the new DST ordinal, increments the entry count in that DST entry, and increments the control point activity count for the request's control point. Now the request is counted twice in the entry/exit counts and in the activity count.

ORDEXDD: all paths rejoin here. Increment the exit count in this 1SP's DST entry. The addition is done so that 7777B is followed by 0001B; MTR increments entry counts the same way. Now the request, if being reissued, is counted just once in the entry/exit counts. Ensure that the DST entry's activity flag is non-zero and mark this request's POOL room empty. If the control point number in the request is zero, go to ORDEXG. Otherwise, issue MTR function M.CCPA (35B, x, x, n, 0) to decrement the activity count for control point n and switch 1SP back to control point zero. Now the request, if being reissued, is counted just once in the control point activity count.

ORDEXG: call .DROPC to release equipment connection and channel reservation. Restore EQUIP byte of DSEWD2. Store (HPOS1) into UNIT+(CUR) and then restore HPOS1 and HPOS2 from the first two bytes of UNIT. Write updated DSEWD2 to second word of DST entry in CM.

ORDEXE: request termination is done. Get first word of DST entry. If entry count = exit count, go to ORDEXF. Otherwise, store difference in CTA and go to Stack Search.

ORDEXF: get PP Job Queue entry count (byte 0 of CM word T.MSC). If it is non-zero, go to ORDEXEA. Otherwise, this 1SP has nothing to do but no

other task needs the PP, so delay 126 microseconds (to avoid causing excessive CM and pyramid activity) and then go back to ORDEXE.

ORDEXEA: look at all DST entries. If only one (this 1SP) has a non-zero activity flag, go back to ORDEXE because this is the only Stack Processor PP and the PP Job Queue does not include another Stack Processor waiting to be assigned to a PP.

ORDEXED: this 1SP has nothing to do, at least one task is waiting to be assigned to a PP, and at least one other 1SP is either running in a PP or waiting to be assigned to a PP. Therefor, this 1SP should release its PP. Rewrite second word of DST entry with activity flag zeroed, restore D.PPIR to its normal value, and release the PP.

9.9.7. Advance Record Block Subroutines

The advance RB subroutines are called when a record block boundary is reached during execution of a request. The three subroutines are: PRUBK for backward positioning, PRURD for read, skip forward, release chain, and rewrite in place operations, and PRUWT for normal writing. Each subroutine returns to its caller with RBR, RBLOC, RBTA, RBTO, and PRU updated, or exits to Termination if necessary to conclude execution of the request or reissue it.

Subroutine PRUBK: called by the O.SKB and O.BPRU executive routines. For O.SKB, a backspace of one PRU must be done unconditionally. A-register negative means this has not been done yet (see SKPBC in 9.9.5.3) so store 7775B in PRU. A-register positive means it has been done (see SKPBA in 9.9.5.3) so store 7776B in PRU. Thus, if PRUBK reissues the request instead of returning to its caller, PRU is set so that when the request is again selected for execution, SKPBAB will know what to do.

PRUBKA: if current position is the first byte (bits 0-2 of RBTO equal bits 0-2 of YBYTE) of the file's first RBT word pair (bits 3-11 of RBTO are zero), report beginning of information.

PRUBKB: find previous record block. Decrement RBTO. If bits 0-2 are now 7, go to PRUBKC to find previous RBT word pair.

PRUBKBA: if byte is not an RB designator, loop to PRUBKA to back up another byte. Otherwise, call SECOMP to setup RBR and RBLOC for this RB and determine its length, and RBRDEV to test for change of device. If new device, go to Termination to reissue the request. Otherwise, set first time switch in .POSIT, set PRU to last PRU of new current RB, and return to caller.

PRUBKC: starting with first word pair in file's RBT chain, read successive word pairs (each points to the next) in the chain until the number of word pairs read is equal to bits 3-11 of RBTO. Go to PRUBKBA.

Subroutine PRURD: called by .POSIT when a new RB must be started during execution of a read, skip forward, or rewrite in place request, and by RCHNZ for release chain requests. If "direct" bit is set (no RBT chain is present), go to PRURDD. If the RB just processed is not to be released, go to PRURDA. Otherwise, call TGLRBR to clear the RBR bit for the RB, and replace the byte in the RBT with the current RBR number rather than zero.

PRURDA: increment RBTO. If the byte just processed was the last of the RBT word pair, get the next word pair or go to PRURDE if there is no next word pair. Call RBRDEV and go to PRURDC1 if the Y-byte now contains an RBR number for a different I/O unit.

PRURDB: if the new RBT byte is zero, loop to PRURDA. If it is an RB designator, go to PRURDX. If it is a new RBR number, call RBRDEV and loop to PRURDA if it references the current I/O unit.

PRURDC1: end of section; i.e., file is continued to another I/O unit.

PRURDE: end of information or end of section. If releasing is not required, go to PRURDF. Otherwise, load 7SZ if not already in core and go to PRURDZ.

PRURDZ: this code releases RBT word pairs from beginning of chain to current position, so that the current position becomes beginning of information. Word pairs are released in groups of eight. Each word pair is inserted into the proper place in the RBT empty chain (pseudo-channel CH.RBT is reserved). If the last 100B words of the RBT area are now all contained in the empty chain, MTR function M.RBTSTO is issued to reduce the size of the RBT area of CM. Return to PRURDF.

PRURDF: if end of section, go to PRURDF1. If end of information, restore RBTO (which may have been made too large by the search for a non-zero byte at PRURDA) to point to the last non-zero byte and go report end of information.

PRURDF1: end of section; reissue the request with new DST ordinal.

PRURDD: direct request; increment the RB designator.

PRURDX: store zero into PRU and return to caller.

Subroutine PRUWT: called by .POSIT when a new record block must be allocated to the file being written. If "direct" bit is set, go to PRUWTD.

PRUWTA: scan forward in current RBT word pair for an empty (zero) byte and go to PRUWTB if one is found. If the word pair is full, call NRBTW to get another word pair, link it to the file's chain, and start in byte 0 of the new word pair.

PRUWTB: call SEEKRB to find an available RB in the current RBR. If one was found, go to PRUTWF. Otherwise, load 7SZ if not in core and go to PRUWTZ.

PRUWTZ: if the current unit is private, go report disk full and reissue the request. Otherwise, call LOCATE to find another suitable RBR. If one is found, go to PRUWTC if it references a different I/O unit or back to PRUWTA if it references the current unit. If none was found, report disk full and reissue the request.

PRUWTC: issue MTR function M.ESTZ (33B, ee, 10B, 0, 0) to set the "active" bit in EST entry ee, and reissue request with new DST ordinal.

PRUWTF: available RB found. Call TGLRBR to set the RBR bit for the RB, store the new RB designator into the current RBT byte, and go to PRUWTX.

PRUWTD: direct request; increment the RB designator.

PRUWTX: store zero into PRU and return to caller.

9.9.8. Miscellaneous Subroutines

This section contains closed subroutines that do not belong in any one of the above sections. Each is called from routines in two or more of the above

sections, and/or from other subroutines in this section. The subroutines are arranged in such a way that each is called only from locations that precede it and/or from 3Sx. Related subroutines are grouped together.

Subroutine MODIFY: used for instruction modification (to set linkage between 1SP and 3Sx and to prepare instructions for reading or writing). Upon entry, the A-register contains the first byte address of a list. The addressed byte contains the first byte address of the address list, and subsequent bytes comprise the value list. The content of each byte of the value list is stored into the cell whose location is contained in the corresponding byte in the address list. The address list is terminated by a zero byte.

Subroutine ASSIGN: set the switch at EREXIT (9.9.6) to show that ASSIGN has been called for the current request. Determine whether 1SP should attach itself to the request's control point (i.e., if the request references a buffer and/or FET or reply word in CM and is not at control point zero). If yes, issue MTR function M.CCPA (35B, x, x, 0, n) to attach 1SP to control point n and store the control point address+W.CPSTAT into CTLPTA; if no, store 0+W.CPSTAT into CTLPTA. Read the CM word addressed by CTLPTA and setup R.CPRA and R.CPFL. If the control point number is not zero and its storage move flag is set, go reissue the request with bypass count set to 3. If the request references an FET but no FST, get the code/status from the first word of the FET. Set abort flag (see EREXIT in 9.9.6) to 0 (don't abort) if the request references an FET and the error processing flag (bit 44 of second word) is 1; otherwise set abort flag to 1 (do abort if unrecoverable error).

Subroutine FETADR: return with A-register negative if the request does not reference an FET or if the FET address is outside the control point's field length; otherwise return with absolute FET address in A-register.

Subroutine SKPEOI: scan RBT chain from current position to end of information, and return with RBT word pair, byte, and PRU numbers pointing to the file's end of information position.

Subroutine NRBTW: obtain a new RBT word pair from the empty chain. If pseudo-channel CH.RBT is not immediately available or the RBT empty

chain has no members, call CALL to call 1SX with code 77B (increase RBT space) and reissue the request with bypass count set to 3. Otherwise, take the empty chain member nearest the high end of CM. If the empty chain is now exhausted, call CALL to call 1SX for more RBT space.

Subroutine GETRBT: read the current RBT word pair from CM into the RBTWRD working storage area. Copy XBYTE into ZBYTE+8 to facilitate end of information test in SECOMP.

Subroutine PUTRBT: write the current RBT word pair, if any, from the RBTWRD working storage area into CM.

Subroutine RBTADR: compute absolute CM address from RBT word pair number. Used by GETRBT and PUTRBT.

Subroutine SECOMP: determine length of current record block and store number of PRUs, minus one, into LASTPRU. First call ADRCOM to setup RBR and RBLOC for the current position. If the RBR has a non-standard allocation style, ADRCOM returns with LASTPRU non-zero; store standardized RB designator into WORKA (where .POSIT expects to find it) and go to SECOMPB. Otherwise, call .SECOMP to get LASTPRU value for a standard RB.

SECOMPB: for writing other than rewrite in place, return to caller. For reading, positioning, and rewrite in place operations, test for end of information. If the <u>next</u> RBT byte is non-zero, return to caller. Otherwise, this is the last RB of the file, so get last RB length (EOIPRU) from first RBT word pair. If (EOIPRU) = 0 or 7777B, go report end of information. If 1 ≤ (EOIPRU) ≤ (LASTPRU), store (EOIPRU)-1 into LASTPRU. If (EOIPRU) = 701B (i.e., file is RESPOND control file or request references no FST entry), or = 7776B (special case for backward positioning), leave RB length-1 in LASTPRU; i.e., assume last RB is full.

Subroutine ADRCOM: setup RBR and RBLOC for current file position. If "direct" bit is set, go to DRW. Otherwise, scan current RBT word pair from beginning to current byte, changing RBR whenever an RBR byte is found. If the current byte is not an RB byte (i.e., rightmost bit is zero), return with zero in RBLOC and in A-register. Otherwise, call GETPOS and return.

DRW: direct request, no RBT from which RBR number can be obtained. Search all RBRs until one is found that references the EST entry specified in the request and has zero in the RB offset byte (START parameter in the RBR macro in CMR). Return with RB designator in RBLOC and A-register.

Subroutine GETPOS: standardize RB designator if necessary. Get RBR header and return with zero in LASTPRU and (RBLOC) in A-register if the RBR has a standard allocation style (00–03). Otherwise, compute the NSA overlay name 7Sx (see 9.2.3) and load it if not already in core. Call 7Sx. The non-standard RB size –1 is in A-register bits 0–11; store this into LASTPRU. The starting PRU number within the corresponding standard RB is in A-register bits 12–17; store this into .ALOPAD (see .SECADR in 9.9.9.1). The standard RB designator is in WORKA; return with this in the A-register.

Subroutine RBRDEV: for positioning, reading, and rewrite in place requests, merely find the EST ordinal, DST ordinal, and unit number for the current RBR. For writing other than rewrite in place requests, RBRDEV also determines whether the current RBR contains available record blocks suitable for allocation to the current file. On entry, the A-register contains an EST ordinal if the file is to be assigned to a specific unit, otherwise zero. On normal return, the A-register contains the logical difference of the unit numbers (bits 0–5) and DST ordinals (bits 6–11) for the RBR and those in DSENUM and CUR, and zeros in bits 12–17. On "unsuitable for writing" return, the A-register contains the complement of the above. In either case, the EST ordinal from the RBR is stored into RBREST. The RBR is considered unsuitable for writing if one or more of the following is true: (1) The RBR is full (available RB count = 0). (2) An EST ordinal was specified and the RBR references another EST ordinal. (3) The "off" bit or "unloaded" bit is set in the EST entry. (4) No EST ordinal was specified and the RBR references a "private" unit or device type = 20B (ECS). (5) The file's allocation style code is not in the list of up to four allocation style codes admitted by this RBR. If the RBR passes all of the above tests and the file has allocation style code 00, assign the primary allocation style code of the RBR to the file.

Subroutine SEEKRB: find an available RB (zero bit) in the current RBR, and return with the RB designator (or zero if none found) in the A-register. If (ALLOC) is 01 or 02 and device type is 6603-I or 6603-II, only record blocks of the specified zone (01 = inner, 02 = outer) are considered. If (RBLOC) is zero, search RBR from beginning to end so that new files tend to congregate in the lowest-numbered RBs; otherwise search the RBR end-around from the position given by RBLOC, so that logically consecutive RBs in a file are physically as close as possible.

Subroutine TGLRBR: invert the RB bit, specified by (RBLOC), in current RBR. Also increment or decrement the available RB count in the RBR header, depending on whether the A-register contains 0 or 1, respectively, upon entry.

Subroutine RBRADR: return with A-register containing the absolute CM address of the current RBR table.

Subroutine SETADC: stores content of A-register (all 18 bits) into the ADC instruction in subroutine ADC.

Subroutine ADC: adds to the A-register the value last stored here by subroutine SETADC. These subroutines are used by SEEKRB and TGLRBR to facilitate access to the current RBR.

Subroutine ERRINF: called by .RDISK after each read or write attempt in which a device (e.g. parity) error is detected. If (ORDEX) = 0000 indicating that an unrecovered error has already occurred during execution of the current request, do nothing. Otherwise store message parameters from SXDATA into the third, fourth, and fifth words of the 1SP message buffer for later use by 1SX when it is called by the Termination section.

Subroutine CALL: call 1SX to another PP. Upon entry, the 1SX error code is in bits 0–5 of the A-register, and bits 6–11 may contain an EST ordinal for disk full codes (10B and 76B). Do nothing if the error code is 10B and the abort flag is zero (don't abort). Report end of information (just to get rid of the request) if error code is 73B–77B and its control point's error flag is non-zero. In all other cases, issue MTR function M.RPJ to call 1SX.

Subroutine CHECKEF: return with A-register = 1 if the current request's control point number is non-zero and that control point's error flag is non-zero, otherwise return with A-register = 0.

Subroutine LOAD7SZ: load overlay 7SZ into the IOBUF area if it is not already there.

9.9.9. Driver Overlays

Section 9.2.2 contains a general description of the driver overlays 3Sx. This section provides further details. The subroutines and tables present in each driver are briefly described below, in the order in which they occur in the assembly listing, followed by a section giving characteristics of each mass storage device type.

9.9.9.1. Subroutines in Each 3Sx

Subroutine .ACTIVC: reserve a channel for accessing the device, and set a switch in subroutine .DROPC indicating that a channel is reserved. For 6000-series devices, only one channel can be used; its number was inserted into all I/O instructions at 1SP initialization time. For 3000-series devices, call R.STB to insert the channel number into all appropriate instructions. For 6638 disk and 865 drums, store angular positions into the ANGPOS array so that .HEAD can include rotational delay (latency) in its access time calculations.

Subroutine .DROPC: if no channel is reserved, do nothing. Otherwise, release equipment connection and reservation if appropriate and drop the channel reservation.

Subroutine .SECADR: given a standard RB designator in A-register, PRU number in PRU, and PRU offset (for non-standard RBs) in .ALOPAD, return with physical sector number in A-register.

Subroutine .HEAD: given target RB designator in A-register, current unit head position or zero in HPOS1, current angular positions in ANGPOS, and target sector number in bits 0-6 of SUBP+(RMNR), return with the

A-register containing a number roughly proportional to the access time for the request. If the current head position is not yet known, HPOS1 contains zero (see OVERA in 9.9.1); in this case, return with an arbitrary value in the A-register.

Subroutine .SHIFT: enter with target RB designator in A-register and current unit head position or zero in HPOS1. Issue hardware functions for head motion (cylinder selection) and/or head group (track) selection as needed. Set first-time switch in .POSIT and store MAXRBCT into RBCOUNT if the installation has made it less than the nominal amount for the device type.

Subroutines .RDINST and .WTINST: setup instructions in 3Sx for reading or writing the mass storage device.

Subroutine .SECOMP: given a standard RB designator in the A-register, return with the number of PRUs in the RB, minus one, in the A-register.

Subroutine .POSIT: if (PRU) ≤ (LASTPRU) and first-time switch is not set, do nothing. If (PRU) > (LASTPRU), call PRURD or PRUWT to advance to the next RB. If the installation has set MAXRBCT to a value less than the nominal amount for the device type, decrement RBCOUNT and reissue the request if the result is negative. If head motion is needed to get from the unit's current position to the new RB, reissue the request so that if the stack contains any requests requiring less access time, they will be executed before the current request is again selected for execution. Otherwise, issue a head group (track) select function if needed and clear first-time switch.

Subroutine .RDISK: transmit one PRU between the IOBUF area and the device, and increment PRU. Call .ERRINF and make retry attempts as necessary if device (e.g. parity) errors are detected. Store 0003 into ORDEX (see 9.9.6) if a retry attempt is successful, unless ORDEX is already 0000. Store 0000 into ORDEX if all attempts fail. This routine handles both reading and writing.

Subroutine .ERRINF: does whatever preparation is needed and calls ERRINF in 1SP (see 9.9.8).

9.9.9.2. Tables in Each 3Sx

Table .CTLR: a list, in R.STB format, of locations of function codes into which the controller equipment number must be inserted (bits 9-11).

Table .CHANT: a list, in R.STB format, of locations of I/O instructions into which the channel number must be inserted (bits 0-5).

Table .PARAM: (generated by the PARAMS macro) a value list, in MODIFY format (see 9.9.8), for which the corresponding address list has been moved from PARAMS to LINK (see OVERA in 9.8.1).

9.9.9.3. Device Characteristics

This section describes the mass storage I/O devices supported by the SCOPE 3 system, and contains specific details of the individual 3Sx driver overlays.

9.9.9.3.1. 6000-Series Devices

The 6000-series mass storage devices are the 6603 and 6638 disk storage units. Since only one channel can be used for each controller, the channel number is inserted at 1SP initialization time and the NSA overlays 7Sx can use all of the space from .CTLR to DRIVEND.

9.9.9.3.1.1. 6603 Disk Storage

The 6603-I is a basic 6603 with or without the first speedup feature (standard option 10098), device type code 01, mnemonic AA, common deck RMSA, driver overlays 3SP and 5CP. The 6603-II is a basic 6603 with both of the speedup features (standard options 10098 and 10124), device type code 04, mnemonic AC, common deck RMSC, driver overlays 3ST and 5CT.

In either case, a 6603 is one controller and one unit. There are 128 cylinders with 8 tracks per cylinder. In each cylinder, tracks 0-3 contain 128 sectors each (outer zone) and tracks 4-7 contain 100 sectors each (inner zone). One PRU = one sector. 322 bytes are written in each 6603-I sector, while 6603-II sectors are 324 bytes. In the 6603-II, the extra two bytes are at the end of

the sector and are used for position verification: the first contains the RB designator and the second contains the sector number. A 6603 record block is a half-track: all of the even-numbered sectors or all of the odd-numbered sectors in a track. Thus there are 2048 RB/unit, of which 1024 have 64 PRU/RB and the others have 50 PRU/RB.

The RB designator is ccc ccc ctt th1

cccccccc = cylinder number
ttt = track number
h = half-track (0 = even, 1 = odd)

and the required sector number is (PRU number) * 2 + h.

In the 1SX error messages, the address bytes are

00c ccc ccc ttt, 000 00s sss sss

ccccccc = cylinder number
ttt = track number
sssssss = sector number

and the status bytes are

xxx rpx xxx xxx, xxx xxx xxx xxx

x = not significant
r = 1 if not ready
p = 1 if parity error.

Bit p is always 1 after a write operation, but does not mean a parity error occurred.

9.9.9.3.1.2. 6638 Disk Storage

Device type code 02, mnemonic AB, common deck RMSB, driver overlays 3SQ and 5CQ. The basic 6638 is one controller with two units, numbered 0 and 1. With the dual access feature (standard option 10037) installed, the 6638 is two controllers, each having one unit numbered 0; they cannot access each other's units. If the option is not installed, card RMSB.20 must be replaced by

```
OPT10037    EQU    0
```

and decks CONTROL and 1SP reassembled, before unit 1 can be used.

Each unit of a 6638 has 32 cylinders, 2 stacks with 16 tracks each per cylinder, and 100 sectors per track. One PRU = one sector. 324 bytes are

written in each sector; the last two bytes contain the physical address in the 1SX error message format (see below) and are used for position verification. A 6638 record block is a half-track: all of the even-numbered sectors or all of the odd-numbered sectors in a track. Thus there are 2048 RB/unit and 50 PRU/RB.

The RB designator is ccc ccs ttt th1

ccccc = cylinder number
s = stack number
tttt = track number
h = half-track (0 = even, 1 = odd)

and the required sector number is

((tttth) * 3 + (PRU number) * 2) modulo 100

which means that each RB starts 3 sectors further around the disk than the preceding RB; this allows time for inter-RB processing (PRURD or PRUWT) without loss of an entire revolution.

In the 1SX error messages, the address bytes are

00c ccc cst ttt, 000 00s sss sss

ccccc = cylinder number
s = stack number
tttt = track number
sssssss = sector number

and the status bytes are

dcr pns sss sss, xaa aaa xcc ccc

d = 1 if lost data
c = 1 if not connected
r = 1 if not ready
p = 1 if parity error
n = stack number
sssssss = sector number
x = not significant
aaaaa = cylinder number, unit 1
ccccc = cylinder number, unit 0

9.9.9.3.2. 3000-Series Devices

The 3000-series mass storage devices are the 854 disk pack units with the 3234 controller and the 865 drum storage units with the 3637B controller. In each case, the controller is attached to the 6000 data channel through the 6681 or 6684 data channel converter. Since these devices can be attached to one or two channels, channel reservation and channel number insertion are performed for each non-immediate request, and NSA overlays can use only the space from .PARAM to DRIVEND. The channel reservation and converter-controller-unit connection are maintained throughout request execution, but the converter is selected only while data, function codes, and status are being transmitted; the converter is deselected the rest of the time so the same channel can be used for data transmission between 1SP and the requestor during PP I/O request execution. The following routines are present only in 3000-series mass storage driver overlays.

Subroutine .PUTADD: load the device address register from ADDREG.

Subroutine .RES: select the converter and connect the controller and unit CUR to it.

Subroutine .DESEL: deselect the converter.

Subroutine .CFR: enter with function code in A-register. Send it out and get controller and converter status. If no error, return with zero in the A-register. If transmission (XMSN) parity error, master clear and return with A-register = 1. Send function code again if controller is busy. If external or internal reject or unit is not ready, return with A-register = 1.

Subroutine .STS: get controller status byte.

9.9.9.3.2.1. 865 Drum Storage

A 3637B controller can have up to eight 865 units, device type code 12B, mnemonic AD, common deck RMSD, driver overlays 3SR and 5CR. In each unit, there are 256 tracks with 128 sectors per track. Each sector is 128 bytes, so one PRU = three consecutive sectors with the last 62 bytes of the third sector being unused. An 865 record block is a half-track: each "even" RB is sectors 0–2, 6–8, ···, 120–122 of a track and each "odd" RB is sectors 3–5, 9–11, ···, 123–125 of a track. Sectors 126 and 127 are not used; this allows time for inter-RB processing (PRURD or PRUWT) without loss of an entire revolution. Thus there are 512 RB/unit and 21 PRU/RB.

The RB designator is   00h ttt ttt tt1

h = half-track (0 = even, 1 = odd)

tttttttt = track number

and the required sector is   (PRU number) * 6 + h * 3.

In the 1SX error messages, the address bytes are

00t ttt ttt tss, sss ss0 000 000

tttttttt = track number

sssssss = sector number (first of PRU)

and the status bytes are

xpx xxx xxx dbr, xxx xxx xxx tcc

x = not significant

p = 1 if parity error

d = 1 if lost data

b = 1 if busy

r = 1 if ready

t = 1 if XMSN parity error

cc = non-zero if internal or external reject.

9.9.9.3.2.2. 854 Disk Pack Unit

A 3234 controller can have up to eight 854 units, device type code 07, mnemonic AP, common deck RMSP, driver overlays 3SS and 5CS. In each unit, there are 200 cylinders, 10 tracks per cylinder, and 16 sectors per track. Each sector is 128 bytes, so one PRU = three consecutive sectors with the last 62 bytes of the third sector being unused. An 854 record block is one track, twice around, as follows:

PRU	sectors
0	0-2
1	6-8
2	12-14
3	3-5
4	9-11

and sector 15 is not used; this allows time for inter-RB processing (PRURD or PRUWT) without loss of an entire revolution.

The RB designator is ttt ttt ttt tt1.

The required cylinder number is (tttttttttt) / 10.

The required track number is (tttttttttt) modulo 10.

The required sector number is ((PRU number) * 6) modulo 15.

In the 1SX error messages, the address bytes are

000 0cc ccc ccc, 000 0tt tts sss

cccccccc = cylinder number

tttt = track number

ssss = sector number (first of PRU)

and the status bytes are

xxx xmx pda ebr, xxx xxx xxx tcc

x = not significant

m = 1 if positioner ready

p = 1 if parity (checkword) error

d = 1 if lost data

a = 1 if address error

e = 1 if any error

b = 1 if busy

r = 1 if ready

t = 1 if XMSN parity error

cc = non-zero if external or internal reject.

9.9.9.3.3. Extended Core Storage

The portion of ECS that is used as a mass storage device has device type code 20B, mnemonic AX, common deck RMSX, and driver overlay 3SX. Details are in section 19.8.0 of this document.

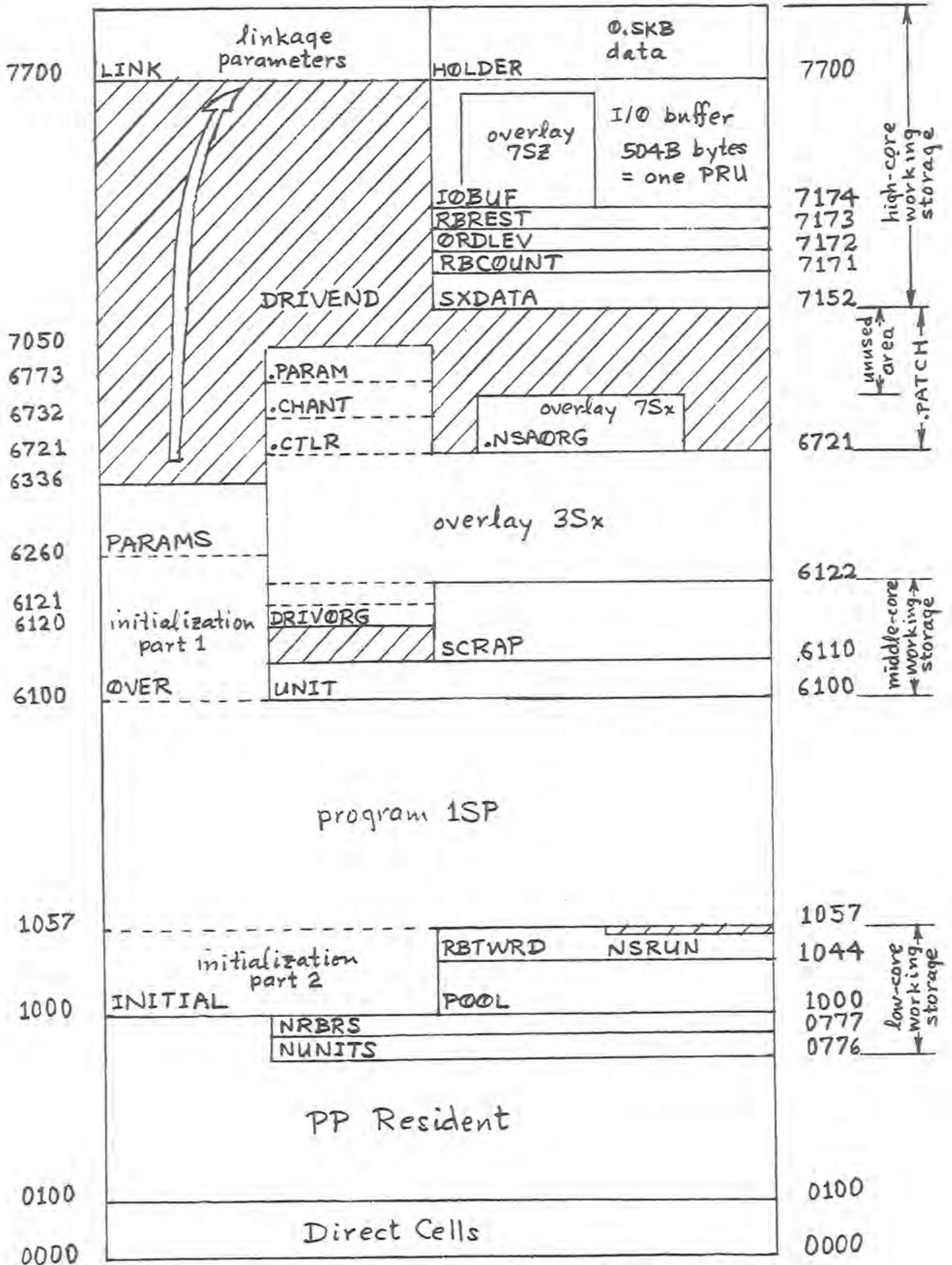


Figure 9-1. Stack Processor PP Memory.

17 - 23 Pointers

17	20	21	22	23
		AFLD IN/OUT for CM I/O RB length for 0.SKB		BFLD OUT/IN for CM I/O HOLDER index for 0.SKB
	6	18	6	18

24 - 30 STLWD Stack Location Word

24	25	26	27	30
STACT no. of word pairs in stack area		STASA fwa/2 of request stack area	NDSE value of N.DEVICE	fwa/8 of DST
	RBR current RBR no.		RBLOC current RB designator	DSESA fwa/2 of DST entry for this 1SP
	9 3			

31 - 35 RBRPWD RBR/RBT Pointer Word

31	32	33	34	35
	RBRFWA fwa of RBR area	RBTE RBT word pair no. of 1st empty chain member	RBTLL RBT area size no. of words/100B	RBTLWA CM size no. of words/100B
		WORKA general scratch		
6	18			

36 - 42 DSEWD1 First Word of DST Entry

36	37	40	41	42
	DRIVNAME "x"	ENCT entry count	CHANNELS second channel / first channel	DSENUM DST ordinal (points to itself)
EMPCT general scratch	RBTOX last valid RBTO (see PRURD)	CXAD stack location of current request	RMNR POOL location of current request	
	6 6		6 6	

43 - 47 DSEWD2 Second Word of DST Entry

43	44	45	46	47
HPOS1 head position unit 0 or unit CUR	HPOS2 head position unit 1	EXCT exit count	EQUIP eq. no. 9 3	active flag
	CUR current unit no.			CSTSWD last code and status
			6	18

Figure 9-2. Stack Processor Direct Cells 17-47.

50–54 First FST Word for request referencing FST
52–56 STEWD First Word of Stack Entry

50	51	52	53		54	55			56			
INDEX	ALLOC	RBTA	RBTO		PRU	ORDER			CPPU			
general scratch	usually, current file's allocation style code	current absolute RBT word pair no.	current relative RBT word pair no.	cur. RB byte no.	current PRU no.	direct	record level	order code	in pool	bypass ct	ctl. pt. no.	DST ordinal
			9	3								

52–61 ANGPOS Current Angular Position of each unit
57–63 Second Word of Stack Entry { STEWDC for CM I/O & Pos. / STEWDP for PP I/O Orders

57		60	61	62		63
FETFWA			START	FLAGS		LIMIT
reply	fwa of FET		fwa of buffer (FIRST) LEVCT record or PRU count	flag bits	lwa+1 of buffer if FET present, otherwise lwa+2	
STPWCT remaining buffer size		STPMBF message buffer address	STPBEG fwa of buffer			STPEND lwa+0 of buffer

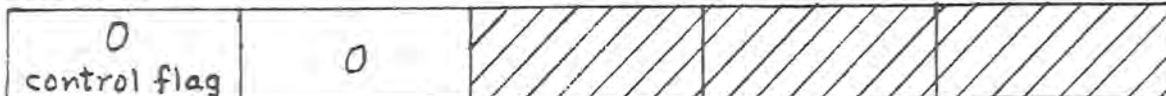
64–70 PPCOMWD Communication Word for PP I/O

64	65	66	67	70
CONFLG control flag	CONWCTI cumulative byte count	CONLWA lwa+1 of data transmitted	CONCHN channel number (control flag = 1)	CONWCTO PRU length (control flag = 4)
			CONSTAT final code and status (control flag = 4)	
	CMWDCT CM buffer part 1 size		CTA CM buffer part 2 size	

71–77 Miscellaneous

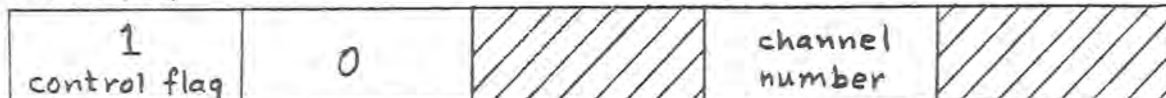
71	72	73	74	75	76	77
TWO	FRBTH	LASTPRU	WDCT	D.PPIR	D.PPOR	D.PPMES1
constant 0002	first RBT address hold	last PRU no., cur. RB	no. of bytes in cur. PRU	input req. address MAXPRU constant 0500B	output register address	fwa of message buffer

Figure 9-3. Stack Processor Direct Cells 50–77.



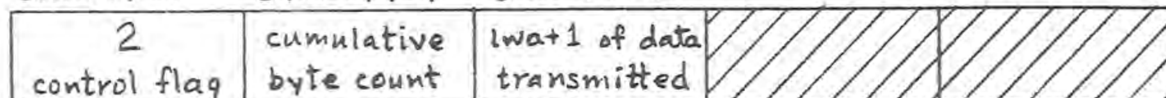
Phase 0: request in stack

Set by requestor when issuing M.EREQS



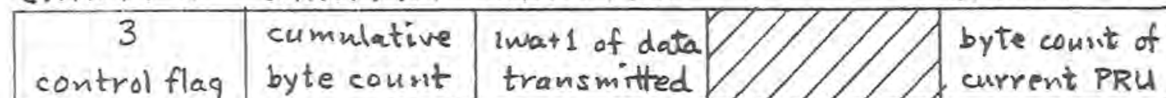
Phase 1: set channel number

Set by 1SP when initiating request execution



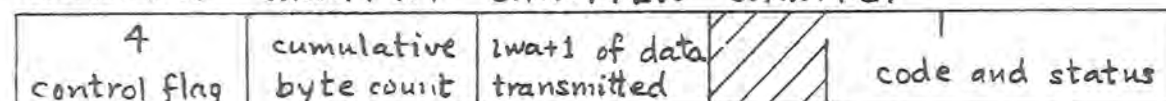
Phase 2: ready for data transmission

Set by requestor after each phase 1 or 3



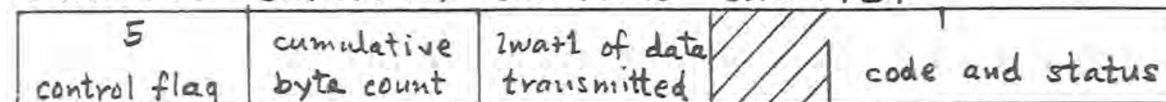
Phase 3: transmission in progress

Set by 1SP after phase 2 if any data to transmit



Phase 4: execution completed

Set by 1SP after phase 2 when no more data



Phase 5: acknowledge completion

Set by requestor after phase 4

Figure 9-4. Communication Word for PP I/O Requests.

9.0 Allocatable Device I/O Program 1SP and its Overlays

(3SP, 3SQ, 3SR, 3XS, 3ST)

The description on the following pages 9-1 through 9-14 is not current. There will be an update in the near future.

9.0 Allocatable Device I/O Program 1SP and its Overlays

(3SP, 3SQ, 3SR, 3XS, 3ST)

The description on the following pages 9-1 through 9-14 is not current. There will be an update in the near future.

1 General Description

There is one common program 1SP and several overlays for each type of allocatable device: 3SP (6603), 3SQ (6638), 3SR (863/865 drum), 3SS (854 disk drive) and 3ST (6603 with option 10124). When 1SP is loaded into a PP, its input register contains the DST ordinal of the assigned device in the last byte and the DST has an index for the appropriate overlay.

The overlays are also used by Dead-Start Loader to drive the device assigned as the system residence.

1SP and a 3SP-type overlay for the system device must be stored in central memory. The rest of the overlays for existing devices may be disk resident but, if so moved, these devices cannot be optional system residences. Those overlays for non-existing devices may be deleted from the system.

If an assembly option is set for HIPRI, a stack request with HIPRI flag will be selected next from the requests for a particular DST. If the definition for HIPRI is deleted (from SCP32.9), the effect of HIPRI flag becomes void.

9.2 Environment

9.2.1 PP Assignment and Release

After a Stack Request (SR) is issued and the entry count in DST is updated through MTR function, MTR checks if a PP is assigned for the appropriate stack processor and, if not, assigns a PP to the DST for stack processing.

Once a PP is assigned to a DST entry for stack processing, the PP will keep processing the SRs for the DST until its exit count becomes equal to its entry count. When they are equal and if there are jobs waiting for PP assignment in the PP job queue, the PP will reset the PP-active byte of the DST to zero to release the PP from the DST.

9.2.2 Other Routines Used, Interfaces and Related Programs

1SX to display error messages,
to abort job if the condition requires, and
to extend RBT empty chain.
7SB overlay for 8-sector RB files.
R.READP/R.WRITEP handles the other end of channel transfers on PP write/read functions.
2ES to format request stack entry.
R.EREQS to add an entry to the request stack.

9.2.3 System Tables and Formats Used

RBR/RBT pointer word, request stack pointer word, T.MSC(PP Job Queue), DST, Request stack, RBR, RBT, FET, FNT/FST and EST

9.2.4 Memory Usage

There are two areas for initialization of 1SP, one at the highest and the other at the lowest part of 1SP.

After the initial loading of 1SP, the portion at the highest is immediately entered to prepare for a 3SP-type overlay (3Sx, x=P,Q,R,S, or T) and to store information necessary for linkage between 1SP and 3Sx in safe area. Then control shifts to the lowest portion to load 3Sx destroying 1SP initialization at high part, and then 3Sx is entered.

3Sx also has two initialization areas one for relocation load (3Sx's use relocatable macros) and the other for linkages between 1SP and 3Sx.

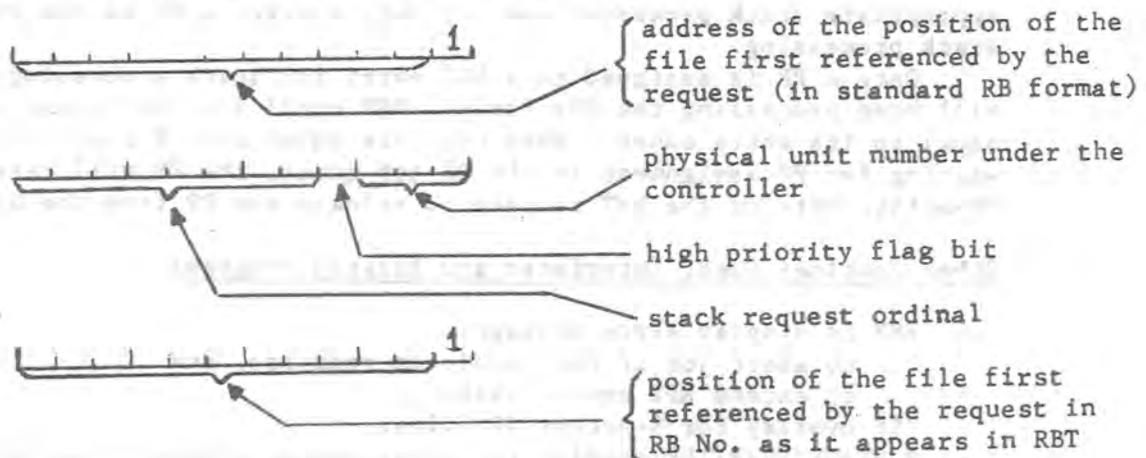
After 3Sx initialization and linkages are made, 1SP initialization at lower part is entered to do initialization for the entire routine and to preset internal tables.

As is seen from the initialization procedure, initialization routines are entirely destroyed after 1SP-3Sx linkage is completed and tables are preset. (see table)

9.2.5 Internal Tables

A) POOL. (C.PPFWA ⟶ C.PPFWA + 3*NRM-1, NRM=12)

12 groups of 3 bytes. Each group is to be assigned to one request waiting for processing. The request stack in CM is surveyed after initial loading of 1SP and after a request is terminated, and, if a request to the current stack processor is newly found, the information about the request is stored into the 3 bytes as a basis for selection of requests to be performed later.

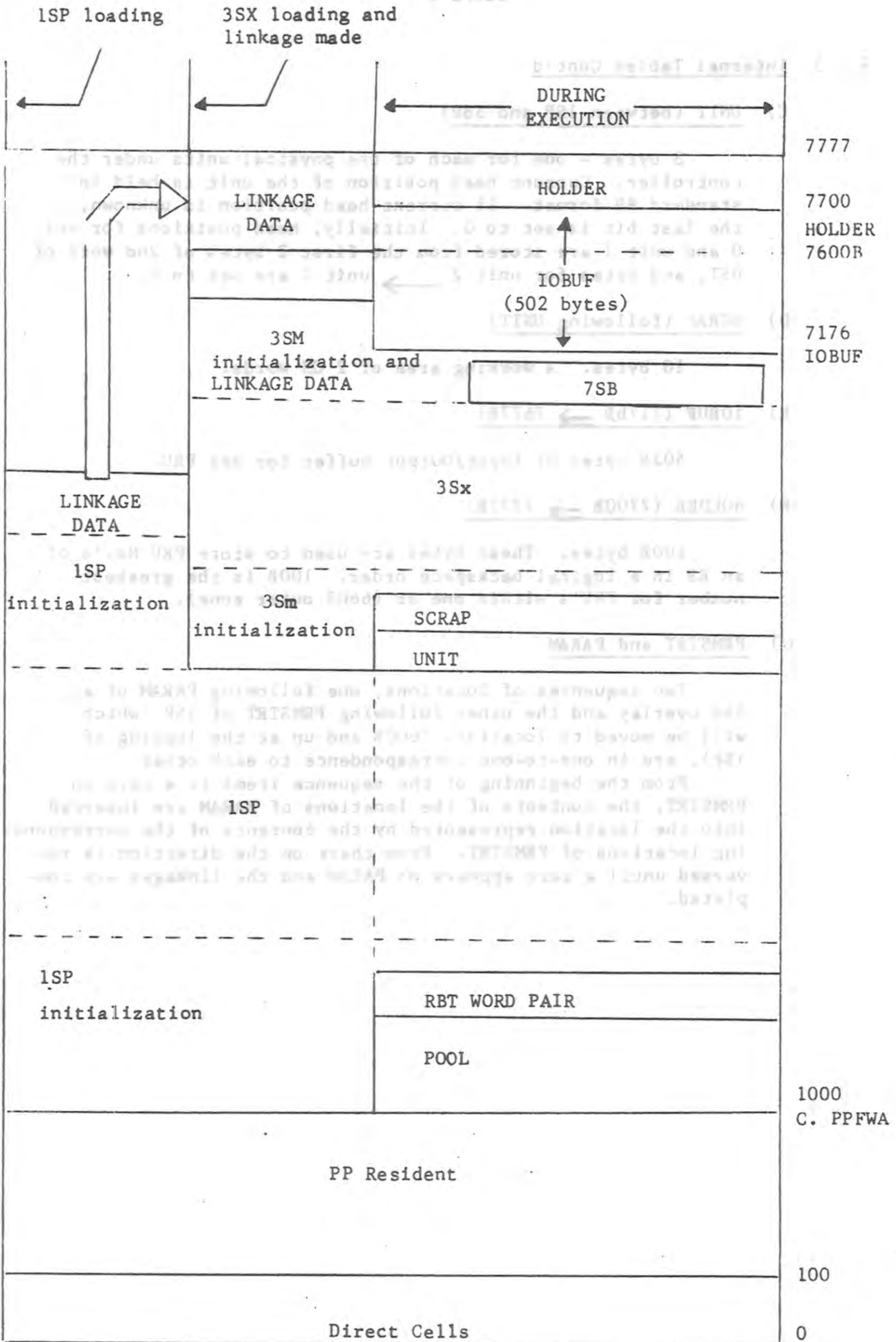


After a request is registered in POOL as stated above, the request in CM is flagged so as not to be selected again. The flag, located in the 11-th bit of first word of a stack request, is dropped when the request is selected for execution.

B) RBT WORD PAIR (C.PPFWA + 3*NRM ⟶RBT WRD)

11 bytes. A working storage for currently referenced RBT word-pair. A copy of first byte (chain address, or 0 if end of chain) is in the 11-th byte.

Memory Usage



9.2.5 Internal Tables Cont'd

C) UNIT (between 1SP and 3Sx)

8 bytes - one for each of the physical units under the controller. Current head position of the unit is held in standard RB format. If current head position is unknown, the last bit is set to 0. Initially, head positions for unit 0 and unit 1 are stored from the first 2 bytes of 2nd word of DST, and bytes for unit 2 → unit 7 are set to 0.

D) SCRAP (following UNIT)

10 bytes. A working area of 2 CM words.

E) IOBUF (7176B → 7677B)

502B bytes of Input/Output buffer for one PRU.

F) HOLDER (7700B → 7777B)

100B bytes. These bytes are used to store PRU No.'s of an RB in a logical backspace order. 100B is the greatest number for PRU's within one RB (6603 outer zone).

G) PRMSTRT and PARAM

Two sequences of locations, one following PARAM of a 3Sx overlay and the other following PRMSTRT of 1SP (which will be moved to locations 7600B and up at the loading of 1SP), are in one-to-one correspondence to each other.

From the beginning of the sequence items to a zero on PRMSTRT, the contents of the locations of PARAM are inserted into the location represented by the contents of the corresponding locations of PRMSTRT. From there on the direction is reversed until a zero appears on PARAM and the linkages are completed.

LINKAGE POINTS BETWEEN 1SP AND 3SP - TYPE OVERLAY

STORED INTO	1SP	3SP, 3ST	3SQ	3SS	3SR
1SP	OVWRTEX+1 SKPFB+1 SKPBB+1 OVWRTR1+1 PPRDA+1 PPWTA+1 RDMSTRA+1 WRMSTRA+1	POSIT +1			
	OVWRTEX+3 SKPFB+3 SKPBB+3 OVWRTR2+1 PPRDA+3 OVPPWT1+1 OVPPWT2+1 OVRDM1+1 OVWRM+1	RDISK +1			RDRUM +1
	OVHED+1	HEAD +1			
	OVSEC+1	SECOMP +1			
	OVRD+1	RDINST +1			
	OVWT+1	WTINST +1			
	SELECTM+1	SHIFT +1			
	OVRDM2 OVRDM2 +1	5500B=RAM DISK1+1		0200B=RJM RESTORE +1	
	OVADRC+1 OVODDB+1	ALOPAD			
	OVPRUB-1	0300B + POSITF-POSINST			
	OVPRUB+1	POSINST +1			
	RDINSTA+1 WTINSTA+1	PRUINST			
	OVCTLR+1	CTLR			
	SELECT+1	ACTIVC+1			
	ORDEXG+1	DROPC+1			
	VERIAB OVVERI	0302B=UJN*+2 0000			0200B=RJM ANGLE +1
	OVSEEK	1203B =LPN 3	0306B =UJN *+6		

Cont'd

STORED INTO	1SP	3SP,3ST	3SQ	3SS	3SR
1SP	OVOV2+1	CHANEND+8+1			
	OVOV2P+1	CHANEND+8+1			
	OVOV1+1	CHANEND+8			
	ODDA+1	CHANEND+8+L.PPHDR+1			
3SP-TYPE OVERLAY	PPIOX	CHANEND			
	PPA	CHANEND+1			
	PCHNA	CHANEND+2			
OVERLAY	PCHNR	CHANEND+3			
	PCHNW	CHANEND+4			
	PCHNC	CHANEND+5			
	PCHNDX	CHANEND+6			
	0000	CHANEND+7			
	SECOMP+1	POSITF+1			
	NUNOFN	POSITC+1			
	ORDEX	RDISKB+1			RDRUMB+1
		RDIS3+1			RDRM3+1
		ERRINFA+1			ERRINFA+1
	ERRCALL		POSITG+1	DISK13+1	RDRUM13+1

9.2.6 Stack Processor Orders

Note: These codes are communicated to the stack processor. They are not the codes used in the code/status word. Order codes requiring a specific request format are indicated. In most cases the format is determined by flag bit settings rather than order code.

O.READ (00): Read into central memory until a short PRU is encountered or the buffer is full(IN=OUT).

O.RDSK (01): Read into central memory until a short PRU is encountered or until the buffer is full. Set the FST to reference the first PRU following the first end-of-record of level x or greater. The level is given in the high-order 6 bits of the order byte. (ie. 1401 would request a read, then would require positioning following an EOR of level 14 or greater)

O.RCMPR (02): Read into central memory after dropping the first three CM words of the first PRU. This is used by STITCH for loading a program from the system library, elimating the three word header added to system programs by EDITLIB.

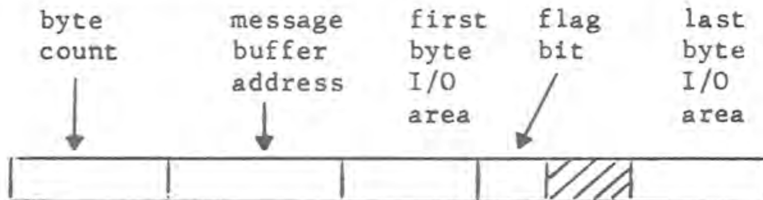
O.RDNS (03): Read into CM non-stop. Stops only upon buffer full, irrecoverable error, zero record, or level 17 logical record.

O.WRT (04): Write full PRU's from central memory.

O.WRTR (05): Write from central memory, ending with a short PRU of the level specified in the high-order six bits of the order byte. If an EOF flag bit is found in this order, a zero length PRU of level 17 will be written following the short PRU terminating the record.

(06,07): Undefined.

O.RDP (10): Read into the requesting PP's memory until a short PRU is encountered, or until the input area is full. For this and other PP I/O orders the format of the second word of the request will be:



O.RDPNP (11): Read into requesting PP after dropping first three CM words of the first PRU. This is used for all PP system program calls.

O.SKP (12): Skip forward n records of level x or greater. The level is specified in the high six bits of the order byte; the number of records to be skipped is given in the third byte of the second word of the order (18 bits). No data is transmitted.

O.SKB (13): Skip backward n records of level x or greater. The level is specified in the high six bits of the order byte; the number of records to be skipped is given in the third byte of the second word of the order (18 bits). No data is transmitted.

O.WRP (14): Write from requesting PP, full PRU's only.

O.WRPR (15): Write from requesting PP, ending with a short PRU of the level specified in the high order six bits of the order byte. If an EOF flag bit is set in this order, a zero length PRU of level 17 will be written following the short PRU terminating the record.

O.BPRU (16): Backspace n PRU's. The number of PRU's to be backspaced is given in the third byte of the second word of the order (18 bits). Note that this requests repositioning defined by physical rather than logical units. No data is transmitted.

O.RCHN (17): Release chain. All record blocks assigned to a file are released, and the RBT word pairs containing them are released. The FST is reset to an empty condition, if its address is supplied in the order.
Note: requests 16 and 17 require no communication with the device, and therefore are given the highest priority in the search for the next order to be executed. All other requests are assigned priority based on the repositioning required.

.3 Equipment Interface

Explanations of the following descriptions

Name of Device (Name of overlay)

RB byte [| | | | | | e | d | c | b | a | 1]
(RB No. is held in RBT word pairs in this FORMAT)

Physical Address [... d | c | b] [... a]

(In the event of device errors current physical address will be output into System Dayfile in this FORMAT. Bit positions with letters correspond to the bit positions of RB with same letters.)

Status Word [... X ...] [...]
STATUS 1 or equipment status — STATUS 2 or 6681 converter status

(In the event of device errors status of the device will be output into System Dayfile in this FORMAT. Bit positions with X represent the conditions that are checked for error after each Input/Output operation.)

6603 (3SP) (also for 6603 with option 10124 (3ST))

RB [← E → | D | C | B | A | 1]

- E: one of 128 head positions
- A: even/odd sectors (0=even)
- B: one of 2 zones of same type
- C: one of 2 stacks
- D: outer/inner zones (0=outer)

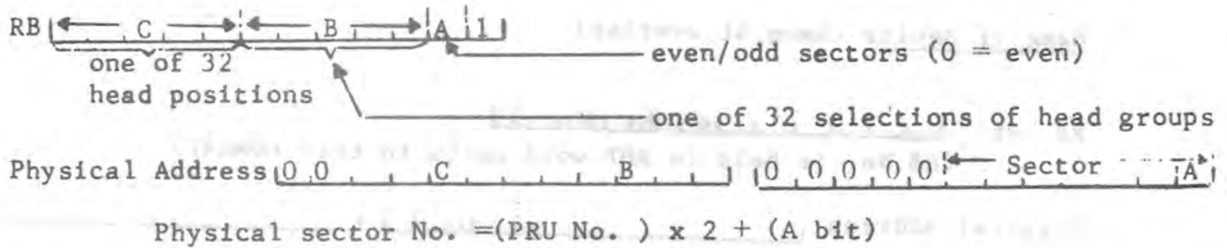
Physical Address [0 | 0 | | | | E | | | | D | C | B] [0 | 0 | 0 | 0 | 0 | ← Sector → | A]

Physical sector No. =(PRU No.) x 2 + (A bit)

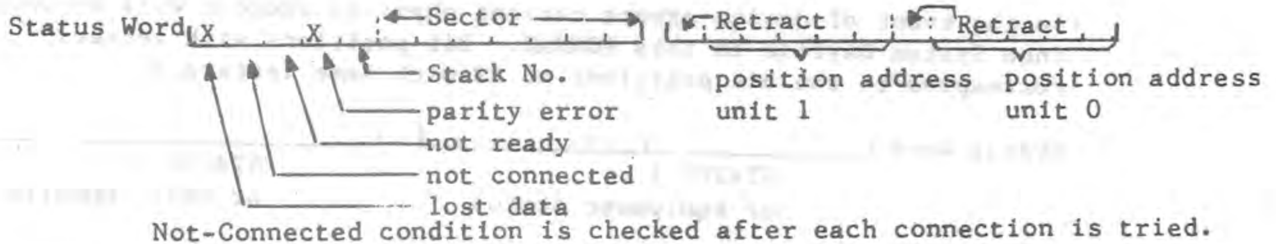
Status Word [... X ...] [none]
- parity error
- not ready

9.3 Equipment Interface Cont'd

6638 (3SQ)

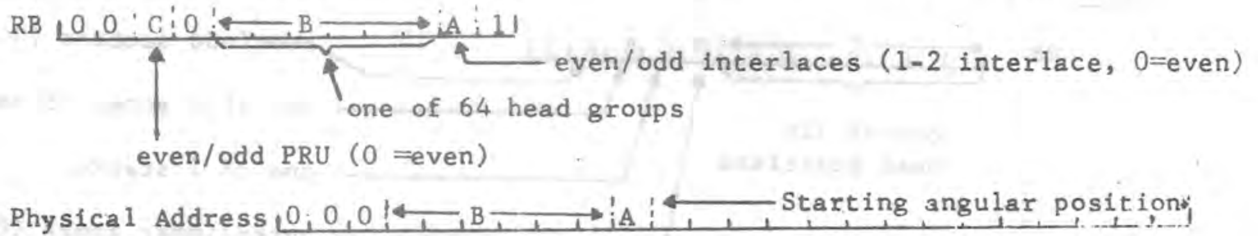


Physical sector No. =(PRU No.) x 2 + (A bit)



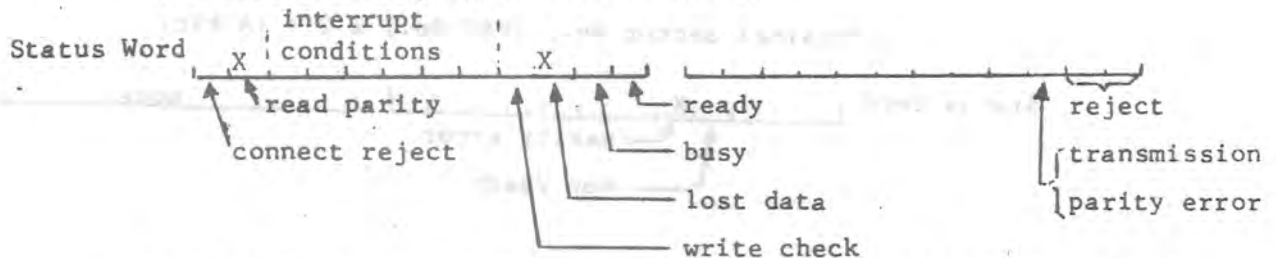
Not-Connected condition is checked after each connection is tried.

863 DRUM under 3637B controller (3SR)



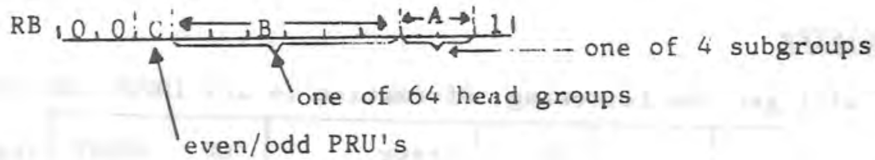
Starting angular position has 7 trailing 0's (16384 byte positions are grouped into 128 "sectors" of 128 bytes)

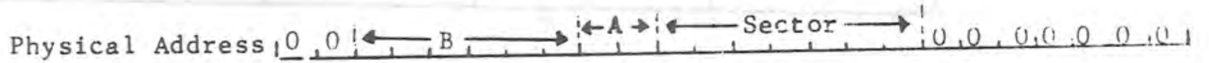
128 sectors are grouped into 42 PRU's.



When a CONNECT or FUNCTION is issued and accepted, READY status is checked.

865 DRUM under 3637 B controller (3SR)

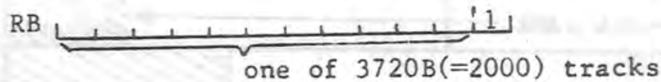


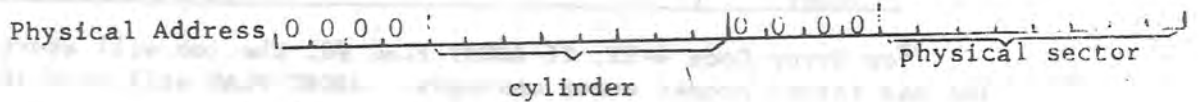


128 sectors are grouped into 42 PRU's.
1 PRU = 3 sectors, 1 sector = 128 bytes.

Status Word (Similar to 863)

854 Disk Drive (3SS)

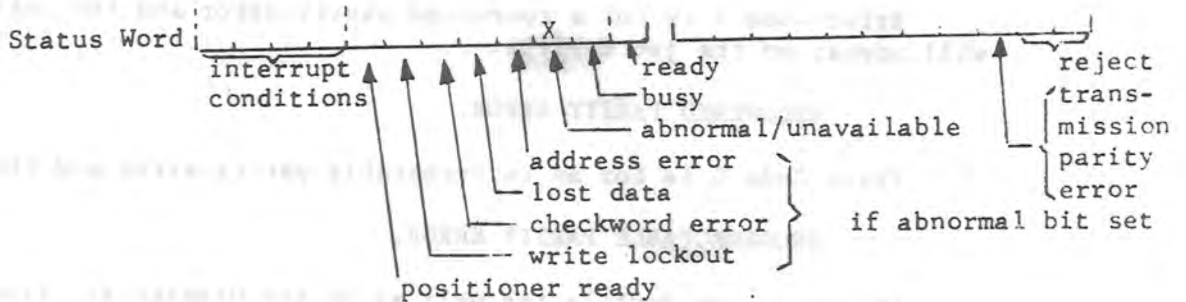




1 cylinder = 10 tracks = 160 physical sectors

1 track = 16 physical sectors = 5 PRU's

1 PRU = 3 sectors, 1 sector = 128 bytes



When a CONNECT or FUNCTION is issued and accepted, READY status is checked.

9.4 1SX

INPUT REGISTER

1SX will get the following information in its INPUT REGISTER:

" 1 S "	" X "	CP / 0 *	Error Code	CP	ABORT FLAG	Stack Processor Mess. Buff. Addr.

*For Error Code 73→77, these 6 bits will be set to 0 as if the request came from CPO. Actual CP No. is always in byte 4.

For Error Code 3,4 and 73, byte 5 has the message buffer address of the Stack Processor PP from which the 1SX request has been issued. Information will be available on the words 3-5 of the buffer, as follows:

word 3	(unused)	RB NO. | 1	RBR NO.	STATUS 1	STATUS 2
word 4	← JOB NAME		→		PRU NO.
word 5	RETRY COUNT	← head position →		(unused)	(unused)

For Error Code 4-22, if ABORT FLAG ≠0, the job will abort after the 1SX has issued proper error messages. ABORT FLAG will be 0 if the error flag in FET (word 2, byte 2, bit 4) is set to 1.

For Error Code 3 (recovered parity error), 1SP will always set the ABORT FLAG to 0.

PARITY ERROR PROCESSING

Error Code 3 is for a recovered parity error and the following message will appear on the job dayfile:

RECOVERED PARITY ERROR.

Error Code 4 is for an uncorrectable parity error and the message is

UNCORRECTABLE PARITY ERROR.

On the system dayfile (as well as on the Display A), along with one of the above messages, information about the location of the error will be displayed:

STATUS XXXX(1) XXXX(2)

EQ XX(3) ADDR XXXX(4) XXXX(5) TRYXX(6)

RBR XX(7) RBXXXX(8) PRU XXX(9)

9.4 1SX Cont'd

PARITY ERROR PROCESSING

(1) Hardware equipment status
(2) Hardware converter or controller status if applicable
(3) EST number of the equipment
(4),(5) Physical starting address of the PRU
(6) Retry count (retry will be made 76B times until it is made to be uncorrectable). Retry Count = 76B for an uncorrectable error.
(7),(8),(9) Logical No.'s for RBR, RB (with flag·bit) and PRU (logical).

The above information will be lost if another parity error has occurred on the same device and has destroyed the information before 1SX picks up the information. In such a case the following message will appear on system dayfile:

PARITY ERROR POSITION LOST.

Note that these PARITY ERROR messages will be used for whatever error conditions occur on a data transfer from the device. Errors include lost data or write-lockout, if applicable. The kind of error can be verified by the octal STATUS bytes in the message.

DEVICE UNAVAILABLE

If a device is not available when it is selected for the execution of a stack request, a request is made for 1SX processing with code 73, and the request is reissued with a delay counter being set.

1SX will find information on the device in the stack processor PP message buffer.

1SX will display the following status message at the control point on the DISPLAY B:

REJECT - XX(1) STATUS XXXX(2) XXXX(3)

(1) EST number of the equipment
(2),(3) the equipment status

As on codes 3 and 4, the information might be lost and the message will be : REJECT - EQUIPMENT NO. LOST.

The message will be kept there for 1.5 seconds and another 1SX will come (with code 74) to clear the message. If the status persists other 1SX's will come with code 73 to keep the message up. A 1SX call with code 73, 76 or 77 will not come until 0.5 seconds after another 1SX call to prevent saturation of the PP job queue.

The condition may occur when the device is not ready, when 6638 connection failed, or when function/connection is rejected because of any system malfunctions.

The operator may drop the control point to get rid of the job in the event of error conditions produced by codes 73, 76, or 77.

RBT EMPTY CHAIN NULL

When 1SP finds that no more empty chain members exist, 1SP will issue a 1SX call with code 77.

If the processing by ISX to get empty chains is not successful, 1SX will issue a message at the control point on Display B: WAITING-RBT STORG, and will issue a delayed call for another 1SX with code 75 or will simply drop itself.

DEVICES FULL

When 1SP finds no more space on appropriate devices available for a stack request, 1SP will issue a 1SX call with code 76.

1SX will issue a message at the control point on Display B:

WAITING - FULL DISK

1SX will issue another 1SX call with a delay of 1.5 seconds to clear the message.

OTHER ERROR CONDITIONS

Codes	Messages
5	NON-EXISTANT RBR REQUESTED
10	BUFFER PARAMETER ERROR
11	BUFFER I/O POINTER ERROR
22	INVALID STACK ENTRY

CHAPTER 10 - TABLE OF CONTENTS

10.0	CHECKPOINT/RESTART	10-1
10.1	CKP - TAPE CHECKPOINT	10-1
10.2	CKP SUBROUTINES	10-2
10.3	RESTART - CHECKPOINTED JOB	10-3
10.4	RST - RESTORE CONTROL POINT AREA FOR RESTART	10-4
10.5	1RC - RELOAD CORE FOR CKP OR RST	10-4
10.6	CY1 - RESET FNT	10-5

10.0 CHECKPOINT/RESTART

10.1 CKP - TAPE CHECKPOINT

Initialization:

Define constants, calculate and store the control point number, parad address, and set a flag if the sp field is non-zero.

Call Validity:

Determine whether the call is an operator or user call. If the call is valid, issue a "CKP REQUEST" dayfile message. If it is a RESPOND or rolled out job, bypass the checkpoint.

Find FNT Entries:

Search the FNT table for entries for this control point. Save the FNT addresses in FNTLIST. Input, output, punch, punchb, LGO, and the memory dump file are always processed. Non-local files are ignored. For each entry, branch to process tape or mass storage.

Process Tapes:

Bypass multi-file tapes. If it is a labelled tape, fetch the label from the FET and save it in LABLIST. Locate CHECKPOINT file if any defined in job.

Process Mass Storage:

Put the FNT address in the last byte of the parameter list entry for the file. Also save the proper copy flag. Generate a parameter list entry if it is not there but is required.

Storage Check:

Increase the storage size if the current storage size is less than the minimum limit required to take a checkpoint.

Memory Dump Process:

If the memory dump file (MDF) has not yet been defined, make an entry in the FNT table. Build an FET for the MDF and open the file. Dump the FL onto the file in the format RA+100B to RA+FL followed by RA to RA+77B.

Define Checkpoint Dump File (CDP):

If the CKP file has not been defined, request one. Open and rewind the file if this is the first checkpoint taken.

Write Initial Header:

Write a 2-word header including the logical file name, checkpoint number, field length, status, and Parad, and a level 16 EOF.

Process Tape List:

Write a file in the CKP tape consisting of a 7-word entry for each tape file. The first three words are the FNT entry and the last four words the label or zeros.

Process ECS:

Bypass if no user area attached to control point. Write a 1-word header consisting of name CCCCCCE and FE/1000B on CKP file. Call EXREAD to read 1000B words from ECS, and copy to CKP file; repeat until whole area has been copied. Follow with EOF.

Process Parameter List:

For every entry in the parameter list write a four-word record and EOF consisting of the list entry and the FNT, and copy the file changing level numbers 17 to 16. For the input file copy both the control card record and data.

Dump CP Areas:

The last file on tape for this checkpoint is the 200B word control point area.

Call overlay:

Call 1RC to reload core from the MDF on disk.

10.2 CKP SUBROUTINES

OPEN

This routine calls CODER to put the OPEN code in the FET for this file, and calls OPE.

CIO

This routine calls CODER to put the code passed in A in the file FET and calls CIO.

CODER

This routine stores 18 bits of A in the FET code/status field. It clears D.T2 - D.T3 and saves the FET address in D.T4 as the PARAD for the call.

RPJ

This routine stores the name, control point number, and auto-recall bit in D.T0 - D.T1 and writes D.T0 - D.T4 into D.PPMES1. It calls monitor to request a PP job and waits until the completion bit is set. If any error occured, a dayfile "I/O ERROR" message is issued.

BUILD

This routine builds the first five words of a FET, given the address of the file FNT.

RESET

This routine sets the word of the file FET preset in AFET2 to the value passed in A.

PAUSE

This routine pauses for storage relocation, resets the RA and FL, and returns if no error has occurred.

COPYS

This routine exits immediately if the copy is not BOI-PP. If the old PRU count is greater than the current count and the EOI bit is not set, return. If it is set, go to copy EOI. If the new PRU count is greater than or equal to the old, write only that part defined by the old PRU count, including EOR's, etc.

WAIT

This routine waits for all activity at the control point to complete.

EXREAD:

This routine calls M.ICE to initiate ICEBOX to read 1000B words from ECS to CM at RA+100B.

10.3 RESTART - CHECKPOINTED JOB

Initialization:

Save the new-field length and the new ECS field length/1000B from the job card. Set up constants and save the parameter count.

Determine Parameters:

Default parameters are LFN=CCCCCCC and checkpoint number = 1.

Locate Proper Checkpoint:

The checkpoint file has checkpoint disposition (0001) if it is not special file name CCCCCCC. Read the checkpoint file until a 2-word header is found. If the name in the header does not match the checkpoint file name supplied on the RESTART card or CCCCCCC by default, pause for the operator to change tapes. Compare the checkpoint number in the header with the requested checkpoint number. Search until a match is found. If the requested checkpoint number is smaller than the located checkpoint number, pause for the operator to back up. If the requested checkpoint number is greater than the largest checkpoint on the tape, issue message to dayfile and abort.

Process Tapes:

For each file, set up an FET and 2 REQUEST parameter words. Request the file, open and rewind it if it is not closed, reposition it, and reset the FNT via CY1.

Process ECS:

Read one word header and check that it contains name CCCCCCE. Compare current FE with old FE to check sufficient ECS attached. Read CKP file until EOF and write to user's ECS area.

Process Mass Storage:

For each mass storage file do the following: Read a 4-word header. If the file name is INPUT, change it to CCCCCCI. Copy the file from the checkpoint tape to mass storage, changing level 16's to 17. Rewind if required. If the file is the memory dump file (CCCCCCM) move the FET to MDFFET at RA+20B. If the file is CCCCCCI, skip one logical record and again call CY1. Read the 200-word control point area and transfer control to RST.

10.4 RST - RESTORE CONTROL POINT AREA FOR RESTART

Initialization:
Set constants, save the control point number, and drop the CPU.

Call Validity:
Check for a valid call by examining the checkpoint file. Find the FNT entry. The file must have checkpoint disposition (0001) or name CCCCCCC. Reset security and code/status fields.

Read Input FST, Header, and CP Area Dump:
The input FST and header are found 2 words ahead of the PARAD address. The input FST replaces word 151B of the control point area, which is the old FST1.

Locate and Process Input Files:
Search for and save the FNT addresses (if any) of the INPUT file and the CCCCCCI file. Interchange names.

Update Control Point Area:
Replace CP area word W.CPFST by dump word W.CPFST. Likewise replace W.CPERT, the control card section, and the auto-recall pointer. Reset the control card pointer in the CP area. Store the checkpoint number in the CP area, word W.CKP.

Update Exchange Package:
Put the current RA, FL and EM in the dump exchange package as well as ECS RA and FL. Then replace the CP area exchange package by the dump exchange package. Call overlay 1RC by giving control to the PP idle program.

10.5 1RC - RELOAD CORE FOR CKP OR RST

1RC is the last phase in the CHECKPOINT/RESTART operation. 1RC reloads from the MEMORY DUMP FILE (MDF) made on the disk by CKP or copied from the checkpoint dump tape to the disk by RESTART. The MDF is made up of field length at checkpoint time in the following format: RA+100B to RA+OLDFL followed by RA to RA+77B.

1RC is called by CKP or RST into the same PP with the following parameter set:

OLDFL	-	must be 300B OLDFL NEWFL
PARAD	-	parameter address; (if less than 100B, special call)
CKPNUM	-	the number of the checkpoint being taken
MDFET	-	at RA+20B there must be a FET for the MDF and the FNT with name CCCCCCM, and equip. type of mass storage.

Processing:

The MDF FET buffer pointer are set to the following: FIRST=IN=OUT=100B, LIMIT=OLDFL. The MDF is rewound and a read is issued to read RA+100B to RA+OLDFL from the MDF into central memory. The RA+100B to RA+277B is saved in PP memory at CMHOLD1+100B. Now a read is issued to read RA to RA+77B from the MDF beginning at RA+100B. Now RA to RA+77B is moved to PP memory at CMHOLD1. The new sense switch settings are set in RA now PP memory.

Releasing of the Disk Storage:

The MDF is then rewound and the disk space is evicted leaving only an FNT for the MDF.

Resetting Core and FNT - FET Linking and Completion Bit:

Now RA to RA+277B is written from PP memory to central memory. Now the old field length (i.e. the user program) has been restored. Next the file name table is searched for files assigned to this control point and the FNT-FET pointers are reset to point to each other. If the checkpoint was not a special call, the completion bit will be set at PARAD.

Ending:

A dayfile message is sent depending on which program called 1RC; that is, CKP or RST. Now the CPU will be restarted and 1RC will drop out of the PP.

10.6 CY1 - RESET FNT

CY1 is called by RESTART to reset the last word of the FST entry of an FNT.

Processing:

CY1 searches the FILE NAME TABLE for a file with the same name as the one given it by RESTART and assigned to this control point. Then the old FST word two is set in the new FST entry. The users label bits from the old FST word one are set in the new FST entry. The new FST word one is given to RESTART. RESTART will use this word in the special case of the INPUT file. The completion bit is set and CY1 drops out.

CHAPTER 11 - TABLE OF CONTENTS

11.0 FILE ACTION REQUESTS 11-1

11.1 OPEN 11-1

- 11.1.1 OPEN Routines and Overlays 11-1
- 11.1.2 OPEN Calls 11-2
- 11.1.3 General Information 11-3
- 11.1.4 Redundant Opens 11-4
- 11.1.5 External Calls 11-4
- 11.1.6 Internal Calls 11-5
- 11.1.7 Multifiles 11-5

11.2 CLOSE 11-7

- 11.2.1 Function 11-7
- 11.2.2 CLOSE File Request 11-8
- 11.2.3 General CLOSE Procedure 11-9
- 11.2.4 CLOSE Reel Procedure 11-9
- 11.2.5 Message Issued by CLO 11-10

11.3 MULTIFILE FNT ENTRY 11-12

11.0 FILE ACTION REQUESTS

11.1 OPEN

11.1.1 OPEN ROUTINES AND OVERLAYS

Open is the file opening procedures for SCOPE 3.0. Its basic purpose is to set files ready for subsequent reading or writing by the various drivers. Open consists of a primary PP routine OPE, and subsequent calls are made to three other routines if necessary:

1MR-open read or open alter on a magnetic tape file
1MW-open write on a magnetic tape file
1MF-file positioning on a multifile labelled tape

OPE only calls 1MR or 1MW, these routines would call 1MF if necessary. The action taken by the open routines is determined by {a} the equipment of the file being opened, {b} the type of open desired, and {c} whether the user has called OPEN or if this is just an internal call from a driver. Overlays to the open routines are based also on the above considerations. High level overlays used by OPEN are 0.

2BP called by OPE on all user calls to OPEN

Link up FNT entry to FET, creates FNT entry if needed.

4ES called by OPE on allocatable devices when-{a} 2BP has set up an FNT entry to assign the proper equipment, {b} to position files on allocatable devices if rewind has been declared, {c} to read the index record on a random file into the user specified index area.

2TB called by 1MR, 1MW and 1MF to position 1/2 inch magnetic tape files {rewind, or skip to beginning of information}

4LB called by 1MR, 1MW and 1MF to read and write labels on a labelled magnetic tape.

Due to the large size of the open routines, some of the above overlays are relocated at a higher level in the PP, for instance, 2BP is loaded at 4000 octal by OPE. An additional call may be made by 1MW or 1MF to load an overlay of the close reel routine {CLO} when an end of reel condition is detected on reading labels when positioning a

multifile or when writing a header label.

11.1.2 OPEN CALLS

The various options of the OPEN call are:

OPEN READ {z=140} - the file is rewound and the security code {bits 18 + 19 of the FNT word 3} is set to read only {01}. If a labelled tape is declared the label is read into the circular buffer and the ≠IN≠ pointer is updated. The PRU count of the file {bytes 4 and 5 of the FNT word 2} is set to zero.

OPEN READNR {z=100} - same as above but the file is not reqound and the PRU count is not affected.

OPEN WRITE {z=144} - the file is rewound and the security code is set to open write {10}. If a labelled tape is declared the VOL and HDR labels are written on the tape and the file is left positioned immediately following the tape mark following the labels. The P.R.U. count in the FNT is set to zero.

OPEN WRITENR {z=104} - the file is not rewound and the security code is set to open write {10}. If a labelled tape is declared the HDR label is written on the tape and the file is positioned immediately following the tape mark after the label.

OPEN ALTER {z=160} - the same action is taken on as described in OPEN READ but the security code is set to open for writing {10.}

OPEN ALTERNR {z=120} - the same action is taken on as described in OPEN READNR but the security code is set to open for writing {10}.

OPEN REEL {z=340} - is generally used by a USER to open subsequent reels on a tape file. This call is also made by the tape drivers at load point of a tape to ensure it is positioned properly, and by CLO when swapping reels atuomatically on a tape file. For reference see the headings of INTERNAL CALL and REDUNDANT OPEN described below. The different driver calls on an internal call define how this call is to be interpreted. A call from close reel on a swap of tape or a user call to open must have the file already opened {i.e. The first tape} in order for OPEN REEL to

determine whether to read or write this file. OPEN REEL therefore, degrades into one of the above specified opens with rewind.

OPEN REELNR {z=300} - is the same as OPEN REEL but the call degrades to an open without rewind.

Certain exceptions are made to the above descriptions for OPENS. If a labelled tape has been declared and a call is made for no rewind and the tape is found to be at load point, label procedures are handled as if the call was specified with rewind. On a random file regardless of the call, the file is rewound after OPEN due to the necessity of referenceing that file to skip to and read the index record. On a labelled multi=file tape, regardless of the call the file is positioned immediately after the labels for that logical file; the rewind option determines the action on the tape, not on the file {see MULTIFILE section}.

11.1.3 General Information

There are certain basic considerations to be kept in mind when using OPEN. One of the primary items that OPEN needs to process a call is the type of device the file is to be {or is} on. This is determined as follows: if the device has been assigned by a RE&UEST function that will be the device used. If the device has not been assigned previously and if no device is specified by the user in his FET, then the device is assigned by 2ES to the most easily accessable allocatable device. If the device has not been assigned previously and the user has specified a device in his FET, this will be the device used. However, the only acceptable FET devices are the allocatable devices, others must be requested and assigned.

FET UPDATING - on a call to OPEN, certain fields are modified or inserted by OPEN and the overlays it calls. If a user calls OPEN, the following fields are altered 0

Device type - replaced by current device of file random bit - turned off if file is not a random file disposition code - replaced by FNT disposition code FNT pointer - replaced by address of FNT entry record blocksize - size depending on device file is on physical record unit size - PRU size depending on device file is on.

On a labelled tape, certain fields in addition to the above mentioned are updated by the open routines and by 4LB, the

label driver. These are filled in only if the fields contain binary zeroes at OPEN time. These are:

Writing
- Edition number - set to 01
- Retention cycle - set to installation standard
- Creation date - set to current date
- Reel number - set to 0001
- Position number - if existing file - correct position Number returned. If new-next available position.

Reading
- File name - from label record
- Retention cycle - from label read- expiration date minus creation
- Creation date - from label read
- Reel number - from label read
- Position number - from first file found matching filename

11.1.4 REDUNDANT OPENS

A redundant open returns a function code 20 to the FET code and status. A redundant open is return on the following conditions:

a. The file has previously been opened {and not closed}
b. The file has never been opened by the user but the PRU

Count in the FNT is ≥ 0 {allocatable devices}

As a redundant open is not considered an error, no message is sent to the dayfile. As a call to OPEN REEL or REELNR is for subsequent calls to a previously opened file, the above conditions are ignored for these calls. Therefore, it is impossible to get a REDUNDANT OPEN on a OPEN REEL call {you can however get an error condition if this call is made on a file which has never been opened}. The FET fields are updated by OPEN even if the call was redundant. On a random file, the index is re-read into the index area and the file is rewound. All other files have no change made to their current position.

11.1.5 EXTERNAL CALLS

User calls to OPEN have the following structure to the input register

BITS	Data
42-59	OPE in display code
40	Automatic Recall
36-38	Control Point Assignment
0-17	FET address

The file action is picked up from the FET code and status bits 3-8.

11.1.6 INTERNAL CALLS

Calls made to OPEN by other PP routines have a different format to the input register. Certain flags are set by these routines to determine subsequent action by OPEN. An internal call is based on the fact that bit 41 is set to 1 (this is protected from the user setting it on by MTR)

BITS	Data
42-59	OPE in display code
41	internal call flag
36-38	control point number
29	on if called by 2TW
28	on if called by 2TR or 2TF
27	on from a previous call from OPE (1MF) - WRITE side
26	on from a previous call from OPE (1MF) - READ SIDE
25	on if called by Checkpoint
12-23	z parameter from OPEN
0 -11	FNT address (first word)

Bit 29 on means that the OPEN REEL call will be handled as an OPEN WRITE, but 28 means it will be handled as an OPEN ALTER. BIT 25 means that there will be no check of old labels before writing on a tape. Bits 26 and 27 are flags set by 1MF when an end of reel was detected spinning for position on a multifile. 1MF will set these and then call CLO to process end of reel, which will call OPEN REEL after it is through. This OPEN will then know this is a special condition of an internal call, will change this call back to a user call, and continue positioning the file.

11.1.7 MULTIFILES

The special processing done by OPEN on multifile tapes merit additional notation. The presence of a multifile is determined by a non-zero value in the multifile name (bits 24-59) of the FET word 13. This field is only checked on a labelled tape file, other types of multifiles can be made by the user but OPEN will handle these as any normal file. For this consideration, a multifile will be defined

as the set of logical labelled tape files consecutively written on a common set of tape(s) and sharing the same 6 digit multifile name. The sequence of these files within the set is determined by a 3 digit position number located in the 11th word of the FET (bits 0-17). If no position is specified for the logical file the correct one will be returned to the user, if there is no existing file the next available position will be assigned to the file (output files only). On a multifile, the reel number in the FET pertains to the number of reels within a logical file, so that the initial reel of a file on the fifth reel of tape of the multi file is one, if the file goes to the sixth reel the FET entry for that file is two. Multifiles use a special FNT master entry to contain common information such as device, physical unit, etc., that is used for each of the logical files within the set. This master entry also contains the current position of the file last referenced by the user (see entry for multifile FNT entry). This entry must be established by the user prior to any action on any of the logical files. This is done by a REWUEST entry of the 6 digit multifile name with a multifile disposition. The user does not have to request each of the logical files, OPEN will create an FNT entry using information from the master entry.

On an OPEN call on a multifile tape, OPEN will position the user to the correct file on that tape. Positioning is determined by the 3 digit FET position number. A check is made prior to positioning to make sure no other files on that tape are still OPENED, so that there will be no possibility of overwriting another file. Processing is different for a file to be opened for reading or writing.

Reading - no positioning will be done if the next record on the tape is a label with the same position number as the FET if there is no position number specified no positioning is done if the filenames are the same (in this case the position number on the label is returned to the FET). Otherwise the file is reset to the last HDR label and positioning continues as described below (A CLOSE NO REWIND and an OPEN NO REWIND on the next file would satisfy the the above conditions).

Writing - no positioning is done if the FET position number is equal to the position number of the last referenced logical file plus one. Otherwise the tape is rewound the VOL label skipped, and positioning continues as described below.

Positioning of a file is done by the overlay 1MF. 1MF is called by the above porcedures, and when complete re-calls the calling overlay. A call to 1MF must imply that a multifile tape already exists and positioning is needed on it. An invalid procedure by the user could result in erroneous processing by 1MF. For instance, a call to OPEN a brand new multifile on the fourth logical file of the set would

result in an attempt to position to a non-existent file. 1MF reads the next HDR label and compares the position number read to the FET. If there is no position number the file-names are compared and if equal the routine exits. If the position numbers compare the routine also exits. If the label position is less than the FET. The tape is spun forward to the next file. If an end-of-reel is found while spinning, a CLOSE-OPEN series of calls continues the positioning on the next reel. If the last file has been read {an EOF label followed by two tape marks} checking is made as described in the write section above, and if these condtions are not met, an illegal function is declared and processing stops. If the position number of the tape is greater than the FET the tape is rewound and the first header on that tape is checked. If the tape is still greater than the fet and the reel number of the label says reel one {any reel other than the first physical reel of the multifile would have reel two in the first header} the job is aborted and a error message is generated- ≠ILLEGAL MULTIFILE POSITION≠. If the reel number is other than reel one, the tape is unloaded and a request is made to the operator to mount the first reel of the multifile set. Processing continues as described above.

Once a user writes a logical file that becomes the last file on that tape. So if a user is writing file position five, decides to rewrite file three, he can no longer reference files four and five unless it is to re-write them {in order}. When no position number is spcified on a file OPEN does not know whether the file identified by the filename already exists or if it is to be a new one. Therefore if a user is writing a series of files without carrying position numbers, he must be aware that each file created will cause a search of the entire set of files. If a user is faced with a problem of not knowing which file is written, a suggestion would be to maintain a counter, update this counter by one and insert it in the next file FET to be written.

11.2 CLOSE

11.2.1 Function

CLO is called to CLOSE files or provide End of Reel processing.

CLO can be called by a user request which is an external call, or by another system program, which is an internal

call. Internal call procedure is used when the file entry location in the FNT is known and marked active. At the completion of internal calls, theFNT and FET will not be marked inactive.

On external calls the address of the associated FET is in bits 0-17 of the input register. The requested CLO function is in the code and status field of the FET. On an internal call bits 0-11 of the input register contain the address of the associated FNT and the requested CLO function is in bits 12-20. Bit 24 is on to indicate an internal call.

The various CLO functions are defined as follows:

130 = CLOSE FILE with no rewind
150 = CLOSE FILE and rewind
170 = CLOSE FILE and unload
330 = CLOSE REEL with no rewind
350 = CLOSE REEL and rewind
370 = CLOSE REEL and unload

11.2.2 CLOSE FILE REQUESTS

File named input or output may not be closed because they have special use in system processing, requests to CLOSE these files are essentially ignored and cause no file action. Processing related to file movement and label handling is determined by device type as follows:

Types 0-37 - allocatable devices

The requested movement, if any, is passed to 2ES which makes the proper stack entry. 2BP is called to monitor the stack request, upon return from 2BP, the stack entry has been completed and the FNT is marked active.

Type 40 = one half inch magnetic tape

If the last operation on the file was not a write or a backspace PRU, the trailer label is read and checked for validity. This check is possible for labelled and unlabelled tapes made under SCOPE 3.1 because trailer label procedure is the same for both types. Unlabelled tapes made external to SCOPE 3.1 need not have valid trailer label. This positioning enables processing of multi-file tapes by issuing CLOSE, no rewind (130), followed by OPEN, no rewind, between Multi-Files. If file movement is requested, 2TB is called to accomplish the movement 2TB will handle trailer label procedure before moving the tape.

are no label considerations.

Type 50 - line printer

Type 60 - card reader

Type 70 - card punch

Any requested file movement is ignored as it is illogical. There are no label considerations.

11.2.3 GENERAL CLOSE PROCEDURE

After movement and label considerations are completed, CLOSE procedure is not oriented to device type. The FNT is marked closed by turning on bits 18 and 19 in FNT (3). If the file is part of a Multi-File group, the MFN FNT entry is marked closed and its LFN pointer is changed to zero. These fields insure proper processing of Multi-File groups and will be updated when the next file in the group is opened.

If CLOSE-UNLOAD (170) is requested, job termination procedure is initiated for the file. The disposition code in the FET is used to determine if the file should be printed or punched. If a print or punch disposition is required, the FNT name is changed to the job name and the FNT entry is assigned to control point zero. If print or punch disposition is not requested, 2DF is called to process the file. 2DF disassociates common files from the control point and writes the index as the last logical record if the file is random. If not common, the file entry is removed from the FNT. If the FNT entry has been deleted, CLO marks the FET inactive and exits. Otherwise, the FNT and FET are marked inactive for external calls. Internal calls exit without completing the FNT or FET, but there are no internal CLOSE file calls in SCOPE 3.0.

If a random file is detected and the security code is either write or alter the index is written out.

11.2.4 CLOSE REEL PROCEDURE

CLOSE reel calls are allowed only for one half inch magnetic tape. Requests for other type devices cause an error condition.

If End-of-Information is outstanding on the reel to be closed, the trailer label is read.

If an End-of-File label (EOF1) is read, the tape is positioned before the tape mark. The End-of-Reel status is

turned off in the FNT, and the normal exit procedure is executed. This process leaves the End-of-Information (EOI) on and the tape positioned so that the tape mark will be detected on subsequent read requests and EOI will continue to be returned.

If an End-of-Volume label (EOV1) is read and the call is internal. the up bit is checked. If the up bit is on, end of reel is returned to the user. The user must then call CLOSE reel and OPEN reel before continuing on the next reel. If the up bit is off, CLOSE will call open to initialize the next reel.

11.2.5 Messages issued by CLO

Pause for OP action

Trailer label is irregular as described by previous message. Operator is requested to type in N.GO, or N.RECHECK. GO will override the irregularity and continue processing. RECHECK will cause the label to be re-read.

MT XX E O T

An end of volume label has been read on an input tape or written on an output tape.

MT XX label unrecognizable

The trailer label on MT XX does not conform to standard EOF1 or EOV1 format.

Redundant CLOSE LFN

The CLOSE request for LFN is redundant as LFN is already closed.

Illegal function LFN

The request on file lfn is not of the type processed by CLO.

Invalid file name lfn

The file lfn has no entry in the fnt and cannot be closed.

INTERNAL (PP) REQUEST TO CLOSE OR OPEN

When file action other than the original user request is desired, the following procedure is used. The system program called to satisfy the original request must call 2BP to validate the FET format and buffer parameters. To secure the associated FNT entry, and to prepare the communication area of PP direct cells. Upon return from 2BP

the associated FNT entry is marked active. Control of this file can be maintained until it is marked inactive in the FNT as this implies action on the file has been completed. Internal call procedure is based on this implication, which enables a chain of file action without releasing the file until all action necessary in satisfying the original request is completed.

When calling a higher level overlay, the requested function is placed in the code and status field of FNT {3}* in direct locations D.FNT+8 and D.FNT+9. The called overlay will determine its processing from the function in the direct cells. Upon return from the called overlay, additional operations may be performed in the same manner or the file may be released. If the file is to be released, the FNT and FET are marked inactive thereby reflecting completion of the original request. This method requires no change in the procedure currently used by I/O drivers as the request is determined from direct locations.

When additional file action requires the full assignment of a PP, the requested function is placed in bits 12-20 of input register of the called PP along with the name of the requested program in bits 42-59. The control point assignment in bits 36-38, bit 41 is on to identify the call as internal, and the FNT CM address is in bits 0-11.

The called program has the responsibility of recognizing the internal call and placing the necessary information in its PP direct cells rather than using 2BP. The system macro PP entry will prepare D.PPIRB, D.RA, D.FL and D.CPAD if D.PPIRB is given as the first parameter of the macro call.

The address of the FNT in the input register is used to read in FNT {2} and FNT {3} into D.FNT and D.FNT+5. FNT {2} address is stored in FA. The relative FET address is used to read FET {1} into D.BA. The appropriate buffer parameters are read from the FET into D.FIRST, D.IN, D.OUT and D.LIMIT. The device type is stored in D.DTS. If the device type is greater than or equal to 40, the EST entry is read into D.EST. This entry is located by the primary device number in FNT {2}.

Upon completion of the requested internal call, the FNT and the FET, are not marked inactive by the called program.

The program which initiated the internal call has the responsibility of marking the associated FET and FNT completed.

---

*third word of the FNT entry.

11.3 MULTIFILE FNT ENTRY

The Multifile FNT entry is used by OPEN and CLOSE to control a multifle tape. It must be initially set up by the user with a REQUEST XXX, MT,MF,N., where XXX represents the six digit multifile identifier. The structure of the FNT entry is 0.

Word		Bits	Description
First word -	Bits	12-14	Control
		17	Lock Bit
		18-59	Filename (left justified)
Second word -	Bits	0-17	Current File position number (display code)
		24-35	Primary device ordinal
		36-47	Secondary device ordinal
		48-49	Tape density
		50	Label bit
		54-59	Equipment code
Third word -	Bits	18-19	this FNT security code (always closed)
		24-35	Multifile disposition (0002)
		36 47	Logical File pointer (FNT Address)
		48-49	Logical File Security Code

CHAPTER 12 - TABLE OF CONTENTS

12.0	GENERAL PURPOSE SYSTEM LOADER (GPSL)	12-1
12.1	PROCESS DESCRIPTION	12-1
	12.1.1 Operation Modes	12-1
	12.1.2 Normal Loading	12-2
	12.1.3 Segment Loading	12-2
	12.1.4 Overlay Loading	12-4
	12.1.5 Memory Maps	12-6
12.2	INTERFACES	12-6
	12.2.1 Internal Interfaces	12-7
	12.2.2 External Interfaces	12-12
12.3	PROCESS FLOW CHART	12-13
12.4	LOD, INITIALIZING LOADER	12-14
	12.4.1 General	12-14
	12.4.2 Detailed Description	12-14
12.5	LDR, CM LOADING AND PROGRAM RELOCATION	12-18
	12.5.1 General	12-18
	12.5.2 Detailed Description	12-22
12.6	LOADER, BOOKKEEPING AND PROGRAM LINKING/DELINKING	12-31
	12.6.1 General	12-31
	12.6.2 Detailed Description	12-33
12.7	CP LOADER	12-68
	12.7.1 Introduction	12-68
	12.7.2 Organization	12-69
	12.7.3 Input - Output	12-72
	12.7.4 Formats	12-74
	12.7.5 LOQ	12-83
	12.7.6 LDQ	12-89
	12.7.7 LOADERQ	12-98
	12.7.8 LOADERS	12-126
	12.7.9 LOADERV	12-128
	12.7.10 LOADERE	12-130
	12.7.11 MAPOUT	12-139

CHAPTER 12 - TABLE OF CONTENTS

12.0 GENERAL PURPOSE SYSTEM LOADER (GPSL) 12-1

12.1 PROCESS DESCRIPTION 12-1

12.1.1 Operation Modes 12-1

12.1.2 Normal Loading 12-2

12.1.3 Segment Loading 12-2

12.1.4 Overlay Loading 12-4

12.1.5 Memory Maps 12-6

12.2 INTERFACES 12-8

12.2.1 Internal Interfaces 12-8

12.2.2 External Interfaces 12-12

12.3 PROCESS FLOW CHART 12-13

12.4 LOD, INITIALIZING LOADER 12-16

12.4.1 General 12-16

12.4.2 Detailed Description 12-16

12.5 LDR, CM LOADING AND PROGRAM RELOCATION 12-18

12.5.1 General 12-18

12.5.2 Detailed Description 12-22

12.6 LOADER, BOOKKEEPING AND PROGRAM LINKING/DELINKING 12-31

12.6.1 General 12-31

12.6.2 Detailed Description 12-33

12.7 CP LOADER 12-68

12.7.1 Introduction 12-68

12.7.2 Organization 12-69

12.7.3 Input - Output 12-72

12.7.4 Formats 12-76

12.7.5 LOO 12-83

12.7.6 LDQ 12-89

12.7.7 LOADERQ 12-98

12.7.8 LOADERS 12-126

12.7.9 LOADREV 12-128

12.7.10 LOADERE 12-130

12.7.11 MAPOUT 12-134

12.0 GENERAL PURPOSE SYSTEM LOADER (GPSL)

See the SCOPE 3 Reference Manual (Section 4) for a description of design concepts.

12.1 PROCESS DESCRIPTION

12.1.1 Operation Modes

The (GPSL) operates in two modes, control card mode and user call mode. The basic difference in the two modes exists in the manner in which GPSL requests are initiated. More specifically, a control card mode is initiated when the operating system detects a control card which requires processing by the GPSL (LOAD, EXECUTE, NOGO, or "program call"); user call mode is initiated when a CPU program requires a GPSL loading operation during execution.

Operating modes are discussed in greater detail below.

(1) CONTROL CARD MODE.

Control cards are interpreted by the operating system program 2TS, TRANSLATE CONTROL STATEMENT. If the control card is a call that requires processing by the GPSL, 2TS calls the GPSL program LOD into the same PPU. Parameters required by the GPSL and the user program are inserted by 2TS in CM locations RA+2 through RA+67 of the user's job area.

The purpose of routine LOD is to insure that the GPSL program LOADER is present in the user's job area. If the LOD call was the result of a LOAD or "program call" control card, LOD will bring the GPSL program LDR into the PPU to immediately perform the requested load. Otherwise, the LOD call will be assumed to have resulted from an EXECUTE or NOGO control card. In this case, LOD requests initiation of the CPU program LOADER and drops the PPU.

LDR accomplishes the input of binary text from the specified file(s). During loading via the PPU memory, address modification for relocation is accomplished by LDR. All LINK, ENTR, FILL, SEGMENT, SECTION and XFER tables are loaded into special points in the job area.

When LOADER determines that a physical load is required, it requests that LDR be loaded into a PPU and executed. LOADER will continue to process in a delay loop until the PPU program is loaded and requests an inactive status for the CPU.

Upon completion of the physical loading, LDR requests MTR to initiate the control point containing LOADER and drops the PPU.

LOADER completes the linking of all interprogram references and places an END in RA+1 to cause MTR to advance to the next control card. The appearance of an EXECUTE card causes the completion of loading. Unsatisfied references are filled by loading appropriate library routines. Residue externals are filled with out-of-bounds references. A memory map is produced, and control is transferred to the specified entry point of the loaded program. The appearance of a NOGO card causes the same action as an EXECUTE card, with the exception that the program load is not executed. Instead, the LOADER tables are cleared, and all pointers re-initiated so that a new load may follow the NOGO card.

Control card formats are specified in SCOPE 3 Reference Manual 3-42.

(2) USER CALL MODE.

The user call mode is essentially identical to the program loading, linking, and initiation functions of the control card mode. Exceptions are that NOGO, EXECUTE and overlay generation functions are restricted to control card mode. Overlay loading is restricted to the user call mode.

The user call is always to the LOADER program which is resident in the user's job area. If the loader is intact, the user's call will be processed. LOADER will call the GPSL program LDR for all physical loading.

Once a program is in operation a user may request the loading of a new binary program or group of programs, overlays or segment. The major difference between overlays and segments is that segmentation permits dynamic storage allocation while overlays are stored in fixed core maps created at initial load time.

The user format is specified in SCOPE 3 Reference Manual.

12.1.2 Normal Loading

Most jobs loaded by the GPSL will not be segmented or overlayed. These jobs will consist of one or more relocatable binary decks, preceded by control cards and followed by data cards.

The binary programs are loaded by placing a LOAD card in the control card deck.

If it is desired to load and immediately execute a major system program, a single "program call" card can be used. To execute major systems programs such as FORTRAN or COMPASS, a card of the following form could be used:

FORTRAN p(1),....p(i)

When the EXECUTE card is encountered the load process will be completed by the GPSL by searching the system libraries in an attempt to fill in all unsatisfied external references. Once this operation is complete, all external references remaining unsatisfied will contain out-of-bounds addresses.

Prior to program execution a memory map is produced and put on the dayfile. CPU program execution will begin at the point indicated in the last loaded XFER table unless another entry point is specified by the EXECUTE card.

12.1.3 Segment Loading

A programmer wishing to segment his programs can initiate loading of the basic segments in a manner similar to that described above. Assuming that the entire program load is contained in the logical record following the control cards, the deck setup might be as follows:

Job Card
Setup Cards
LOAD (INPUT)
EXECUTE
$^{7}8_{9}$ (end-of-record)

SEGZERO (P, A, B, C)
SEGMENT (G, D, E, F)
(prog A deck)
(prog B deck)
(prog C deck)
(prog D deck)
(prog E deck)
(prog F deck)
$^{7}8_{9}$ (end-of-record)

Data Decks
$6_{7_{8_{9}}}$ (end-of-file)

The LOAD card will cause a load from the input file; however, unlike the normal loading scheme, the SEGZERO card triggers only the actual load of the segment zero programs (A,B,C). All loading of segments at levels greater than zero must be accomplished by the user in his object code.

The execution (during object program operation) of a user call initiating the load of a segment at a level lower than existing segments causes a process of "delinking." The GPSL delinks all segments equal to or less than the level of the called segment. All programs to remain in core must have external references which point to entries in the segment being de-linked re-established as unsatisfied externals. This entails filling such addresses with out-of-bounds addresses and clearing certain positions of the LINK tables.

The deck setup example included at the beginning of this section will now be extended to illustrate several features of segment loading. Assume a user call of the following parameter list format is processed for user call format.

Fn = INPUT
SL = pointer to a list with the following program names: E,F, and A
L = 1.

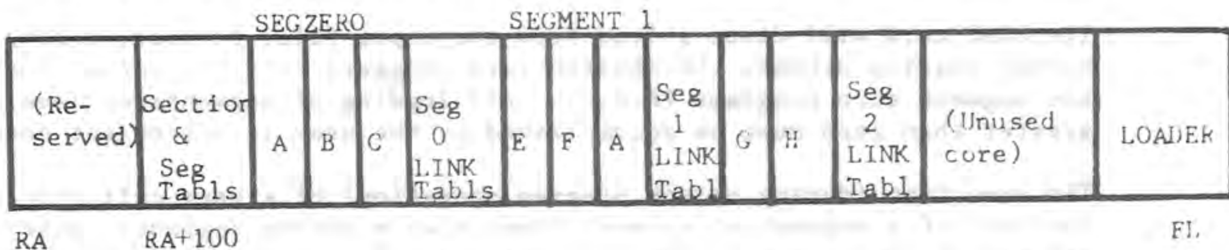
Note that program A was also loaded in SEGZERO. It is permitted to load the same program at several segment levels. If this case exists, however, loading proceeds in the following manner: All programs in this segment (level 1) are loaded and linked to each other first. Any unsatisfied externals from other segments are then linked to the entry points in the segment. Since program A already exists at a lower level, all external references to A occurring in lower level segments will have been already satisfied. Thus, no program in a lower level segment may reference a program in a higher level segment if a copy of that program already exists at a lower level. A core map after the loading of segment levels 0 and 1 would appear as follows:

		SEGZERO				SEGMENT 1					
(Re-served)	Section & Segment Tables	A	B	C	Seg 0 LINK Tables	E	F	A	Seg 1 LINK Tabl	Unused core	LOADER
RA	RA+100	RA+M									FL

A user call might also load programs not appearing on the INPUT file. In this case the file name Fn in the parameter list must not be zero and all programs in that segment must come from file Fn. For example, assume that later in the execution of the object program we have been discussing a user call with the following parameter list format is processed:

Fn Q
SL pointer to a list with the following program names: G,H
L 2

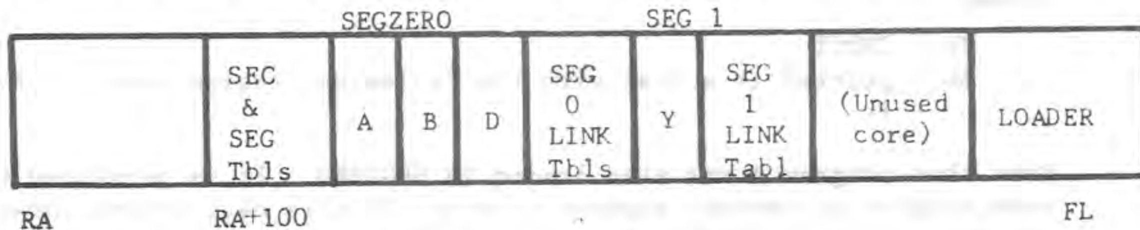
Another segment will be loaded (and linked) beyond the first two segments at level 2. This segment would be loaded entirely from file Q. A core map of this load would appear as follows:



As a further example, assume a subsequent call with the following parameter list format is processed:

Fn = X
SL = pointer to a list with the following program name: Y
L = 1

Current segments levels 1 and 2 will be delinked and the new segment loaded at level 1. A core map after this action would appear as follows:



12.1.4 Overlay Loading

Overlay loading differs from segmentation loading in that all overlays are relocated, linked and written to a "save" file in absolute format prior to being called at execution time.

In addition, certain other features are unique to the overlay loading scheme. They are:

- Shrinkage of blank common may be accomplished during loading by origining an overlay at a location within blank common.
- The deck structure for overlay loading is similar to that for segment loading. In the overlay structure, however, a BCD OVERLAY loader directive must precede each program or program group to be included in that level. The first overlay level must be (0,0).

If the user wishes to build and load overlays, a deck setup of the following type might be used:

```
Job Card
Setup Cards
LOAD (INPUT)
EXECUTE
7
 8
  9   (end-of-record)
OVERLAY (Q, 0, 0)
(Prog A Deck)
(Prog B Deck)
OVERLAY (1, 0, C000200)
(Prog C Deck)
OVERLAY (1, 1)
(Prog D Deck)
(Prog E Deck)
OVERLAY (1, 2)
(Prog F Deck)
(Prog G Deck)
OVERLAY (2, 0)
(Prog H Deck)
OVERLAY (2, 1)
(Prog I Deck)
7
 8
  9   (end-of-record)
Data Decks
6
 7
  8
   9  (end-of-file)
```

The GPSL will read in all decks for OVERLAY (0,0), in this case programs A and B. All linking will be done and loading completed by searching the library and filling in external references. A special version of LOADER, called OVERLOD , is attached to the upper end of OVERLAY (0,0) for loading of overlays at execution time. The largest blank COMMON reference, if present, will be origined immediately following OVERLAY (0,0). For the current example, assume that a 2000 word block of blank COMMON is origined immediately following OVERLAY (0,0).

Once the building of OVERLAY (0,0) is completed, it is written onto the specified file. The next program, OVERLAY (1,0), is then read into memory in the same position vacated by OVERLAY (0,0). However, "virtual" address relocation is performed on the program text (in the current example, the base address used for relocation would be the origin of blank COMMON + 200). Again, the overlay is written to the "save" file.

Subsequent overlay decks are processed in the same manner and written to the "save" file. Consequently, the file would have the following format following the overlay generation phase:

O V L 0 0	Prog A	Prog B	O V E R L O D	O V L 1, 0	Prog C	O V L 1, 1	Prog D	Prog E	O V L 1, 2	Prog F	Prog G	O V L 2, 0	Prog H	O V L 2, 1	Prog I	EOF

If an EXECUTE card is processed next, execution will begin at the entry point for OVERLAY (0,0). Subsequent overlays are called by the user in his object program - only OVERLAY (0,0) is loaded as a result of a control card. A typical user call would be:

Fn = X (X = name of "save" file)
SL = 0
L_1 = 1
L_2 = 0
V = 1

In this case, OVERLAY (1,0) would be called from the "save" file. Following the loading process, the overlay entry address is placed in word 2 of the call. Assume a user call from OVERLAY (0,0) of the format:

Fn = X
SL = 0
L_1 = 2
L_2 = 0
V = 1

In this case, OVERLAY (2,0) would be called into memory immediately following blank COMMON.

As a final example, assume that an OVERLAY (1,0) was called by either OVERLAY (1,1) or OVERLAY (2,0). Such a call is illegal and will result in: (1) the loading of all overlays and recording of consequent memory maps; and (2) the bypassing of the next EXECUTE card (this does not necessarily mean the abortion of the entire job).

12.1.5 Memory Maps

A short list of data is placed on the output file each time a load is completed. In addition, every time an overlay is generated the loader will place on the output file certain information identifying the overlay and the location in which it is to be loaded.

This output is provided to facilitate debugging and testing and may be suppressed by the proper setting of the NOMAP flag in the user call parameter list.

During segment loading a map similar to the standard load map is produced for each load. This may be suppressed by properly setting the NOMAP flag. When loading from control cards, a map is always produced except for system programs (such as FORTRAN and COMPASS) loaded from the library.

12.2 INTERFACES

The GPSL programs communicate with each other and the remainder of the SCOPE system through a series of parameters (status indications and pointers) located in CM. Parameters utilized for communication between GPSL programs are termed internal and those utilized for communication between GPSL programs and other SCOPE system programs are termed external.

Two general CM areas are used for program communication.

(1) location RA through RA+77_8 of the user's job area.

(2) the system pointers, tables, and communication areas in low CM core.

The low core composition is covered in other descriptions of the SCOPE system and will not be discussed in detail in the GPSL documentation.

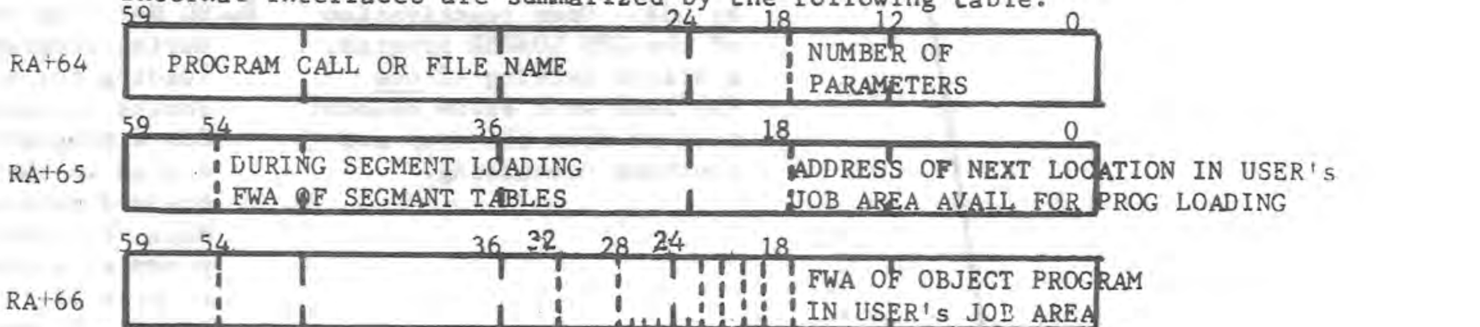
12.2.1 Internal Interfaces

The GPSL programs communicate with each other through four words in CM: RA+64 through 67 of the user's job area and words W.PPTIME and W.CPLDR in the jobs control point area.

Parameter data contained in each of the CM words are referenced as bit configuration within a 60 bit CM word of the following format:

CM Word:

Internal interfaces are summarized by the following table.



Address of the next location available for the loading of tables accompanying LOADER in CM. As the tables are loaded adjacent to and below LOADER, successive locations for loading are obtained by negative addressing from this pointer.

FLAGS SET BY LOD to indicate the following options:

bit 32 REDUCE for current run

bits 33-35 001=MAP(ON)
010=MAP(OFF)
100=MAP(PART)

FLAGS SET BY 2TS when the following cards occur:

bit 28 DEBUG - LOADER will write DEBUG file so that DMP may produce labeled dumps.

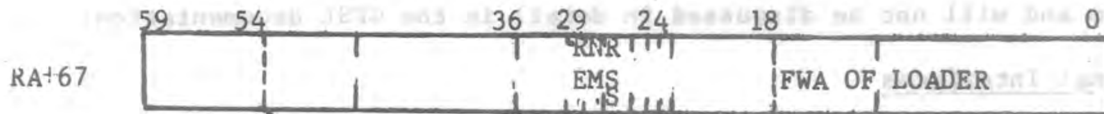
bit 29 DEBUG(C) LOADER will write DEBUG file so that DMP may produce labeled and change dumps.

bit 30 DEBUG with T parameter-During overlay or segmentation runs, this bit indicates to LOADER if TRACE is being used.

bit 31 DEBUG with S parameter-During overlay or segmentation runs, this bit indicates to LOADER if SNAP is being used.

LOADER DIRECTIVE CARD FLAGS: SET WHEN LDR DETECTS THE CARDS
SECTION(bit18)
SEGZERO(bit19)
OVERLAY(bit20)
SEGMENT(bit21)

OVERLAY FLAG BITE giving the level of the incoming overlay



During segment loading the address of the linkage table accompanying the lowest level program in user's job area

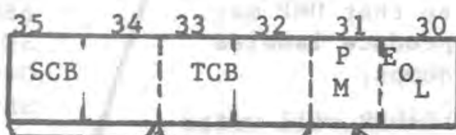
See below

the last control card type interpreted by 2TS:
000="PROGRAM CALL" card
001=LOAD CARD
010=EXECUTE card
100=NOGO card
(External interface)

RSS mode operation

REQUEST exit flag LOADER routine REQUEST sets this bit to zero before exciting to LDR to service a loading request. REQUEST will be in a loop monitoring the bit when the CPU is dropped by LDR. Upon reactivation of the CPU LOADER program, a status setting of one (by LDR) will allow REQUEST to exit from the loop and continue processing.

NO MAP flag set if, during program loading for a job, access is requested for a program not listed in the library directory. Maps will then be produced according to bits 33-35 in RA+66. No memory maps are produced for runs whose pr grams are loaded entirely from th system libraries.



END OF LOAD bit: indicates to LOADER that LDR has completed the requested load.

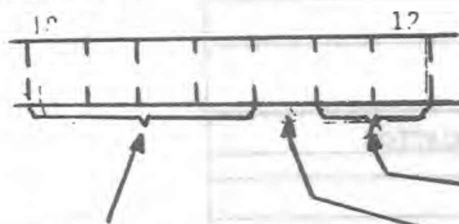
PARTIAL MAP bit: indicates that a single line header is to be output instead of the entire map.

SNAP control bits set by LOD:
01=SNAP cards but not TRACE cards have occurred.
10=SNAP cards have been followed by TRACE cards, thus making any more SNAP cards illegal.

TRACE control bits set by LOD:
01=TRACE cards but not SNAP cards have occurred.
10=TRACE cards have been followed by SNAP cards, thus making any more TRACE cards illegal.

Control Point + 24 Bit 36 "Loader already in" flag: indicates that LOADER has been loaded the the user's job area (by a prior access to the LOD program).

Control Point W.CPLDR



Set by 2TS when DEBUG card is processed. LOD places these bits in RA+66 (bits 28-31) when initializing for each run.

Set by 2TS when REDUCE card is processed. LOD places this in RA+B6 (bit 32) when initializing for each run.

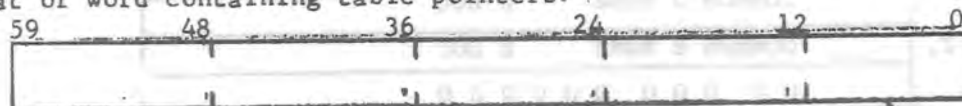
Set by 2TS when MAP card is processed.
- 00=MAP (OFF)
- 01=MAP (ON)
- 10=MAP (PART)

LOD formats these bits into RA+66 (bits 33-35) when initializing for each run.

Program linkage is accomplished by LOADER through the use of a set of tables formatted and loaded by LDR. These tables are adjacent to the LOADER routine in the user's job area. A core image after a normal link and load would appear as follows on the next page.

NOTES for table on page 12-10

1. Format of word containing table pointers.



Position of LINK table relative to start of PTBL NO.1

Position of ENTRY point table relative to start of PTBL NO.1

Position of FILL table relative to start of PTBL NO.1

Position of REPL table relative to start of PTBL NO.1

Position of 1st word of PTBL NO.2 relative to start of PTBL NO.1

2. Following the above word is a list of COMMON block names referenced by this program. Blank common is indicated by bits 54-59=77. A zero word terminates the list.

3. Each pointer is relative to the start of this PTBL.

4. During segment processing tables for each program are condensed and moved to a position adjacent to the program in lower core. The contents of this position will be changed to contain a pointer to the set of linkage tables adjacent to the next higher level in core. The pointer will be set 0 for the last set of tables in the list.

		Memory	
		L O A D E R	RA + FL
		XFER NAME XFERADD	
		POINTER TO BLANK COMMON	
		PREVIOUS XFER	
PTBL #1.	1.	PROGRAM #1 NAME PROG. LOCATION	
		COMMON A NAME A LOC	
		COMMON B NAME B LOC	
	2.	COMMON C NAME C LOC	
		0 0 000 000000	
		LINK TABLE	
		0 0 000 001	
		ENTRY POINT TABLE	
		FILL TABLE	
		ENTRY POINT TABLE	
		REPL TABLE	
	3.	PROGRAM #2 NAME PROG. LOCATION	
		COMMON D NAME D LOC	
	2.	COMMON E NAME E LOC	
		0 0 000 000000	
PTBL # 2		ENTRY POINT TABLE	
		LINK TABLE	
		REPL TABLE	
		FILL TABLE	
		U N U S E D C O R E	
		PROGRAM #2	
		COMMON E	
		COMMON D	
		PROGRAM #1	
		COMMON C	
		COMMON B	
		COMMON A	
		SEGMENT TABLE	
		SECTION TABLE	RA + 100

The tables included in the core image diagram of the previous page are outlined in greater detail below.

SECTION

SECTION NAME 1 BIT 59=1	0 0 0 0
PROGRAM NAME 1	0
PROGRAM NAME 2 BIT 59=1	0 0 0 0
SECTION NAME 2 BIT 59=1	0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	

*FOR LAST PROGRAM IN LIST

SEGMENT

SEGMENT NAME 1 BIT 59=1	0 0 0 0 0
PROG. OR SECTION NAME 1	0
PROG. OR SECTION NAME 2	0 0 0 0
SEGMENT NAME 2 BIT 59=1	

OVERLAY

50_8	L2	L1	FWA ENTRY ADDRESS

ENTRY POINT

0 0 0 0 0 0	NEXT ENTRY TABLE
ENTRY POINT 1 NAME	ABS LOCATION
1RA	ADDRESS
ENTRY POINT 2 NAME	ABS LOCATION
1RL	ADDRESS
ENTRY POINT 3 NAME	ABS LOCATION
1RL	ADDRESS
0 0 0 0 0 0 0 0 0 0 0	

LINK

0 0 0 0 0 0 0	NEXT ENTRY TABLE
EXTERNAL REF. 1	0 IF UNSATIS FIED OR ADDRESS
P RL LOC EXTERNAL REF- ERENCE *2 014 UNSAT P RL LOC	
0 0 0 0 0 0 0 0 0	

REPL

0 0 0 0 0 0 0		NEXT REPL
I	SR	S
C B	DR	D
0 0 0 0 0 0 0 0 0		

FILL

0 0 0 0 0 0 0		NEXT FILL
CONTROL BYTE BIT 59=0	RL	ADDRESS
RL ADDRESS	0 0 0 0 0 0	
0 0 0 0 0 0 0 0 0		

12.2.2 External Interfaces

External interfaces are summarized by the following table. The "program" column refers to the SCOPE system program with which the GPSL interfaces.

Program	Description
2TS	The system program 2TS will call LOD as a PPU overlay after processing a LOAD, NOGO, EXECUTE or "program call" control card. The control card parameters will be placed in the user's job area starting at RA +2. In the "program call" case where no identifier is found, the first word on the card will be interpreted as a parameter and stored in RA +64. In addition, 2TS will store a code in RA +67, bits 24-26, indicating the type of control card interpreted. Refer to table of RA +67 for the applicable codes. PPU program loading is now done by LOD.
MTR	MTR or storage cells set by MTR are utilized by the GPSL in the following ways.

1. The PPU programs LOD and LDR will interpret the program by which they were called by the contents of their associated PP input register.

The conditions indicated by the contents of bit positions 41-59 (in display code format) are as follows:

CONTENTS	CONDITION
1AJ	LOD called by 2TS or, LDR called by LOD
LDR	LDR called by LOADER
LOD	LOD called by LOADER (bits 0 17=0)
LOD	LOD called by RUN (bits 0 17=64b)

2. LOADER will call the PPU programs in the following manner:

LDR or LOD will be put in RA+1, left justified and in display code format. In the case of a user call, the address of the associated parameter string will also be placed in RA+1, right justified. LOADER will continue to execute in a delay loop until the CPU is dropped by the called PPU program.

3. The PPU programs may enter the CPU program LOADER at several different entry points. The entry point is preset by altering the program address in word 0 of the associated control point. The PPU

then requests the CPU with a MTR function code of 15.

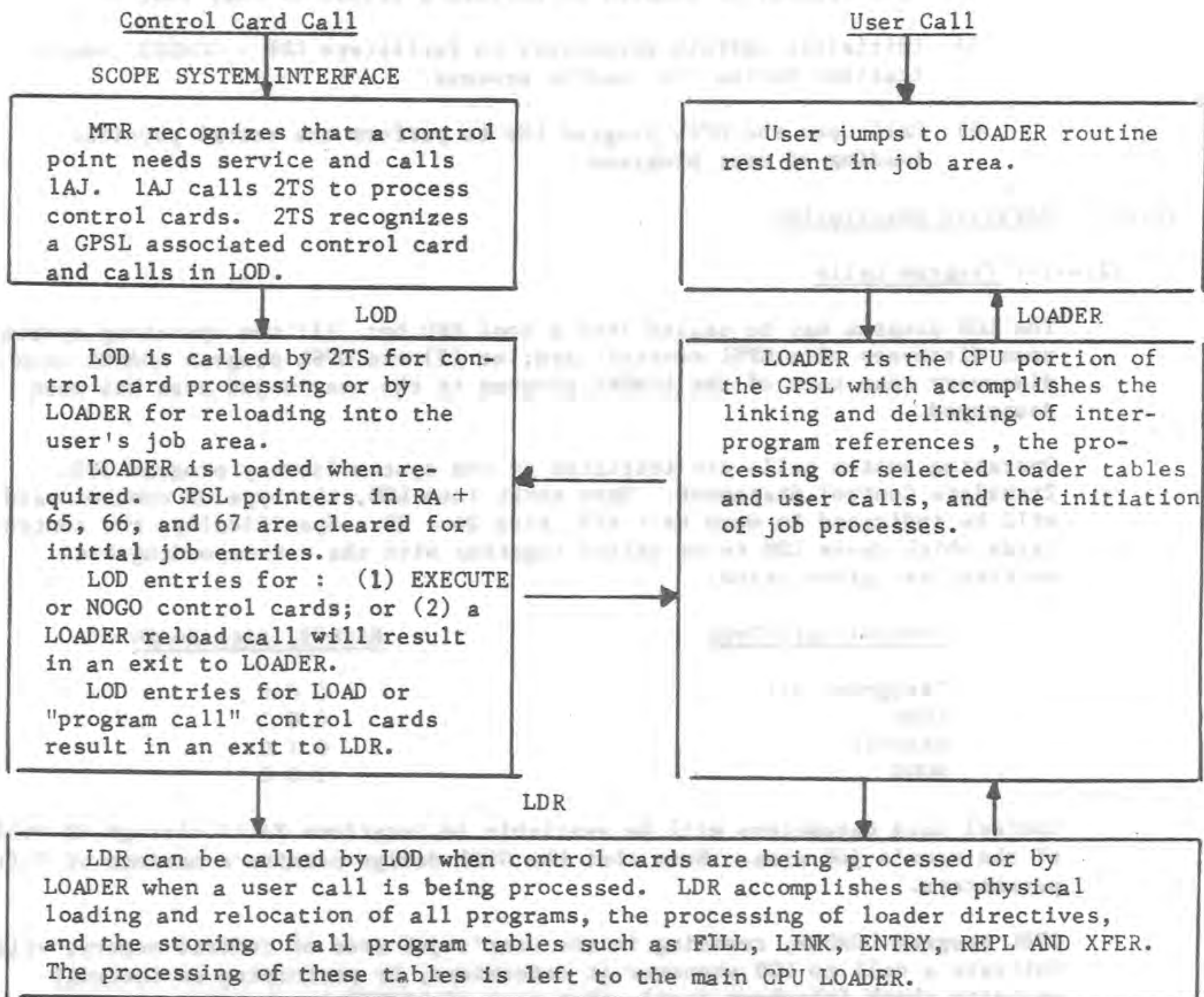
4. LOADER may request the system to access the next control card by placing an END in RA+1.

5. The LDR routine will access word 23 of the user's control point area at the end of each program load in order that load times may be included in user requested memory maps.

CLL The system program CLL willbbe dropped from SCOPE system library, and its functions performed by LDR.

CIO The system program CIO will be utilized to output memory maps and OVERLAY.

12.3 PROCESS FLOW CHART



12.4 LOD, INITIALIZING LOADER

12.4.1 General

The three primary purposes of the GPSL program LOD, initializing Loader are: (1) to perform the initial loading of the GPSL routine LOADER into the user's job area, (2) to ensure that the routine LOADER is intact in the user's job area upon subsequent calls, and (3) to perform certain initializing tasks in order to facilitate the loading process.

More specifically, the following main functions are performed by the LOD program:

1) Perform the initial loading of LOADER upon encountering the initial GPSL control card
2) Test for the presence of LOADER in central memory by performing a preliminary checksum test for the validity of LOADER
3) Upon failure of the checksum test, to perform a reload of LOADER
4) Upon request of LOADER, to perform a reload of that routine
5) Initialize certain parameters to facilitate LDR - LOADER communications during the loading process
6) Call upon the GPSL program LDR to perform the actual physical loading of user programs

12.4.2 Detailed Description

12.4.2.1 Program Calls

The LOD program may be called into a pool PPU by: (1) the operating system upon discovery of a GPSL control card; or (2) the GPSL program LOADER upon discovery that part of the LOADER program in the user's job area has been destroyed.

Operating system calls are initiated by the system library program 2TS, Translate Control Statement. Upon entry into LOD, the type of control card will be indicated in word RA + 67B, bits 24 - 26. Specifically, the control cards which cause LOD to be called together with the corresponding bit settings are given below:

Control Card Type	RA+67B, Bits 24-26
"Program Call"	0 0 0
LOAD	0 0 1
EXECUTE	0 1 0
NOGO	1 0 0

Control card parameters will be available in locations RA +2 through RA +63B of the user's job area. Note that the GPSL design permits a maximum of fifty parameters.

GPSL program LOADER, residing in the user's job area of central memory, will initiate a call to LOD whenever it determines, by performing an internal validity check (checksum test), that part of LOADER has been destroyed. The LOD program will reload LOADER into central memory and return control of the CPU to an entry point within the LOADER validity test routine.

12.4.2.2 LOADER Residence Test

When the LOD is called by LOADER, it is presumed that part of LOADER has been destroyed and a reload of LOADER ensues. On the other hand, when the LOD program is called by 2TS, it is possible that the LOADER routine has already been loaded by a previous request. This condition ("LOADER-Already-In") may be determined by checking a status bit in the user's control point area --- the LOADER -Already-In Flag. Specifically, this status flag occupies bit 24 of the control point area plus 24B.

If the status bit indicates that LOADER is already resident in the user's job area (i.e. equal to 1), LOD will perform a preliminary validity (checksum) test for a portion of LOADER. A valid result for this test will result in the bypassing of the LOD process which loads the LOADER program. LOD does not perform a total checksum of LOADER due to the excessive processing time that would be required. Specifically, the portion of LOADER that is checksummed by LOD is the routine within LOADER which performs the checksum of the entirety of LOADER and which calls LOD if that checksum fails.

12.4.2.3 Pointer Initialization

If the LOADER-Already-In Flag indicates that this is the first entry into the LOD program for the current run (i.e. equal to 0), all fields in the LOADER communication area in RA +65B through RA +67B will be initiated as follows:

1) CORNEXT, the next available location for program loading is set to 100B (RA + 65B, bit 0 - 17).

2) TBLNEXT, the next available location for storing loader tables is set to FWA LOADER - 4 (RA +66B, bits 36-54).

3) The control card type bits, RA + 67B, bits 24 - 26, which are set set by 2TS, and the RSS bit, RA + 67B (bit 27), which is set by LOD, are preserved.

4) FWA LOADER in RA + 67B, bits 0 - 17, is preserved.

5) The MAP, REDUCE, and DEBUG control bits are picked up from the control point area and placed in RA + 66, bits 28 - 35. The REDUCE BIT in the control point area is cleared.

6) The remaining bits in RA + 65B through RA + 67B are cleared.

7) Words FWALODR-1 through FWALODR-5 are cleared.

Should the LOD program call be the result of an EXECUTE, NOGO, SNAP, TRACE, or "program call" control card, the LOADER-Already-In Flag will be set to 0 prior to program exit. This will cause GPSL initialization --- reload of LOADER and initialization of address pointers --- upon the processing of a subsequent control card for the job in question.

12.4.2.4 LOADER Loading Process

The process required to load the LOADER program into central memory consists of two major functions: (1) Searching the system library to determine the location of the LOADER program, and (2) the physical loading and relocating

of LOADER into the user's central memory job area.

The library directory is searched for the presence of LOADER. Failure to locate the program results in the error message "LOADER NOT FOUND IN LIBRARY" and termination of the job.

If LOADER is found to be on disk resident, then a disk READ request is issued which will read as much of the LOADER text stream as possible into the CM area which LOADER will occupy in its final state. Since the text stream is longer than the relocated program, the remainder of the stream is kept in the PPU. This method is used to ensure that the disk will be read at maximum speed and no disk revolutions lost. Once this process has been accomplished, the unrelocated text is processed one word at a time and replaced in central memory.

If LOADER is found in the RSL, then the text stream is processed one word at a time from its location in CM resident.

The LOD input routine accomplishes the loading and relocating of the LOADER program into the extreme end of the user's job area. Upon termination of the loading process, LOD will place in word RA + 67B, bits 0 - 17, a pointer to the FWA of LOADER.

12.4.2.5 Program Exits

There are four possible exits from LOD.

1) Error Exit -- the following error conditions will cause the job to be aborted with the appropriate message:

   A) LOADER NOT FOUND IN LIBRARY -- LOD could not find an entry in the Program Name Table for LOADER.

   B) FL TOO SMALL FOR LOADER - The user's field length is not large enough to contain all of LOADER.

   C) INVALID CONTROL CARD - The name specified on a program call card cannot be located in either the FNT or the Library Directory, or there is a format error in the parameters on a control card requesting a PP program.

   D) LOADER CONTROL CARD OUT OF SEQUENCE - An EXECUTE or NOGO card has appeared without a prior LOAD card pertaining to this run.

   E) ERROR IN LOADING LOADER - Internal checks on the stack processor performance while reading LOADER from disk have failed. One of the following conditions is the cause:

      1. A false End-of-Record status is given after the start of LOADER is read to Central Memory.

      2. End-of-Record status does not occur after reading the remainder of LOADER into PP memory.

   F) TRACE OR SNAP CARDS NOT CONTIGUOUS - while processing SNAP and TRACE cards, LOD has discovered that the sequence SNAP-TRACE-SNAP or TRACE-

SNAP-TRACE has occurred. All cards of one type must appear before the cards of the other type.

2) PP program exit -- When a program call control card specified the name of a PP program, LOD requests that the program be loaded into another PP and then exits to the idle loop.

3) LDR Exit.

If the LOD program call was the result of a LOAD or "program call" control card, the GPSL program LDR is called into the PPU to begin loading of user program text immediately.

4) LOADER Exit.

If the LOD program call was the result of a LOADER call from the CPU, or an EXECUTE or NOGO control card, MTR is requested to activate the LOADER program in the CPU. The PPU is dropped by LOD. A program call card for SNAP or TRACE will also cause an immediate exit to LOADER if the appropriate routine (SNAP or TRACE) is already loaded as a result of a previous SNAP or TRACE card.

Prior to normal exits (LDR and LOADER) from LOD, the LOADER entry address for subsequent processing is preset. This is accomplished by setting the program address in the related control point exchange package to the proper entry point. The LOADER entry point settings are summarized by the following table. See Section 12.4 LOADER, for a more detailed description of LOADER entry points.

Entry Point	Location	Condition
LOADER 1	RA + FL - 2	Return used by LOD after it has completed the reload of LOADER upon the request of LOADER
LOADER 2	RA + FL - 3	LOADER has just been loaded initially or just reloaded. Upon entry into LOADER, an initial checksum must be taken for subsequent comparison.
LOADER 3	RA + FL - 4	LOADER was not reloaded, as it has passed the preliminary checksum test. Upon re-entry into LOADER, a checksum will be taken and compared with the value stored when LOADER was initially loaded.

LOADER 4	RA + FL - 5	Overlaps are being generated and this is at least the second LOAD card for this run. The checksum is verified upon entry to LOADER.
LOADER 5	RA + FL - 6	This is the entry to LOADER when instant replication is to be performed. This entry is made from LDR, but is mentioned here for sake of continuity.

12.4.2.6 PP Call Processing

When the LOD program is called by 2TS as a result of a control card, it is immediately discernible to LOD what type of control card is involved. This can be determined by examination of RA + 67B, bits 24-26. However, it cannot be determined in the case of a "program call" control card whether the program calls for the execution of a CPU or PPU program. Hence, when this type of call to LOD is made, LOD searches the File Name Table (FNT), the Library Program Name Table and the Library Entry Point Name Table.

If a match is found in any of these three lists, LOD assumes that a call for a CPU program has been made and proceeds to load LOADER according to the process described above.

However, if a program name cannot be found in any of these three lists, LOD presumes that a call for a PPU program has been initiated. In this case LOD performs certain preliminary checks on the validity of that assumption. Specifically, these checks are: (1) the program name must not exceed three characters in length, (2) the first character must be alphabetic, (3) there must be no more than two parameters present, and (4) the parameters must consist entirely of octal digits. If the program call passes these tests, LOD forms a PP call word from the name together with the parameters and places this word in the PP recall register associated with the control point in question. If the program call is for DMP, checks (3) and (4) above are skipped, and the PP call word is formatted with bits 0-35 set to all ones. This gives DMP the capability of accepting more than two parameters and accepting alphanumeric parameters, both of which are needed for labeled dumps. The PP containing LOD is then released. If the program call does not pass the above tests, the message "INVALID CONTROL CARD" is issued to the dayfile, and the control point is aborted.

12.5 LDR, CM LOADING AND PROGRAM RELOCATION

12.5.1 General

12.5.1.1 Introduction

LDR is the GPSL routine which accomplishes the physical loading and relocation of programs into the user's central memory job area. The routine resides in the system peripheral library (RPL or PLD) and may be called by GPSL routine LOD to process control card calls or GPSL routine LOADER to process user calls.

Input text or data to be processed by LDR must be in relocatable subroutine, loader directive or absolute overlay format. Relocatable subroutine format

consists of a group of tables within a logical record which contain the information necessary to load, relocate, and link the subprogram. Loader directives are control cards which accompany program text and supply information to the GPSL regarding the building and loading of jobs containing overlays or segments. Absolute overlays represent a special class of subprograms which have been preprocessed by the GPSL and are no longer in relocatable subroutine format. Although such programs cannot be relocated and linked at load time, substantial savings are afforded the user in the amount of time required for program loading and execution.

12.5.1.2 Program Interfaces

The loading of all user programs is provided by the combination of the GPSL PPU routine LDR and the GPSL CM routine LOADER. The major functions of LDR in the loading process are as follows:

1) Formatting and storing of relocatable subroutine format tables (PIDL, FILL, LINK, ENTR, REPL, and XFER) for subsequent LOADER processing.

2) Processing and relocation of program addresses for relocatable subroutine format TEXT tables.

3) Editing of loader directives and formatting of SECTION and SEGMENT tables for subsequent LOADER processing.

4) Absolute overlay generation and loading.

5) File and library searching and selective program loading.

6) Error processing for invalid program formats, loader directives and/or file composition.

7) Physical loading of programs from disk and magnetic tape I/O devices.

12.5.1.3 Program Composition

Three PPU routines (2LA, 2LB, 2LE) are retained on the system library as overlays to LDR. Overlay 2LA is called to edit loader directives and, in certain processing modes, load absolute overlays; overlay 2LB performs no processing and is used to restore LDR after 2LA has been called and terminated; and overlay 2LE is utilized to format and output error conditions detected by LDR and 2LA. Overlay routines 2LA, 2LB, and 2LE are described in greater detail in Section 12.5.2.

LDR and the program overlays communicate through a series of program vectors and constants at the extreme lower end of the LDR program. As these locations are absolutely addressed by the overlay routines, it is mandatory that their direct core locations not be altered.

The upper end of PP core occupied by LDR is used as the input buffer during normal relocatable loading. Depending on the PRU size of the input device (disk, disk pack, drum, etc.) several portions of the LDR program which are required only during the initial LDR entry or are not required during the normal loading operation are located in the buffer area and will be overlayed when triple buffer loading is being performed.

To the extent practical, LDR is constructed in a modular fashion with distinct blocks of coding performing functions which may be logically grouped together.

The modules composing the LDR program are described in greater detail in Section 12.5.2.

12.5.1.4 LDR Calls

As previously stated, LDR is called by GPSL routine LOD as an overlay to process control card calls and by GPSL routine LOADER to process user calls. Error conditions detected in the control card mode will result in the abortion of the job. Error conditions detected in the user call mode will normally result in the abortion of the loading operation and a return to the user with an error flag denoting the nature of the error (fatal or non-fatal). When such an error occurs during the loading from a file, the integrity of loads from the same file by subsequent user calls cannot be guaranteed.

When LDR is called upon to load from a file, the system file directories FNT and library directory are searched for the file (or program) name in the designated order. Absence of the file (or program) name in the aforementioned directories will result in the output of an error diagnostic prior to job abortion or user return.

12.5.1.5 Loader Directive Processing

Loader directives are processed by LDR only in the control card mode and are ignored during user call processing. SECTION and SEGMENT directives are edited and formatted into tables beginning at RA + 100 of the user's job area. When such directives are encountered at beginning of a file, loading does not proceed in the normal manner. After formatting the SECTION and SEGMENT tables, LDR will exit to LOADER prior to the loading of any program text. LOADER will return to LDR with a call to load only those programs from the file which comprise the initial segment (SEGZERO). Subsequent control card calls (occurring prior to the execution of the current job) which request loading from files containing SECTION and SEGMENT cards will result in the loader directives being ignored. Within the same job, control card calls requesting the loading of files containing both overlay and segment loader directives will result in the abortion of the job.

A control card call requesting the loading of a file containing overlay loader directives will result in the conversion of all programs on the file to the absolute overlay format. LDR will build overlay tables within LOADER from the overlay loader directive and load each program on the file sequentially. LOADER will process each program in the normal relocatable subroutine load manner and write it to the file(s) designated by the loader directives. When all programs on the file have been processed, the control card will be analyzed for its type. If the type was a "program call", the first (0,0) level overlay generated will be loaded into the user's job area and executed at the entry point. If a LOAD control card was processed, control will be returned to the operating system to process the next control card.

12.5.1.6 Normal Relocatable Loading

Programs in relocatable subroutine format consist of PIDL, ENTR, TEXT, FILL, REPL, LINK, and XFER tables as described in the ref manual. With the exception of the PIDL, which must precede all tables for a given program, tables may be encountered in any order during the loading of a program.

TEXT tables are loaded directly into their specific locations by LDR and all relocation of addresses (other than that directed by FILL tables) accomplished.

Relocation of programs after they have once been relocated into CM from the specified file is not permitted, as all relocation bits are discarded. A program to be relocated to some other core area must be reloaded from the applicable file.

All remaining tables are formatted in tables descending through the user's job area and immediately below LOADER. These tables are used by LOADER for program linking and delinking. Sufficient rocm must be allowed in the job area for the program, LOADER, and all tables. If, during the loading process, LDR detects that LOADER tables and program text are being overrun, the load or the job (depending on the type of call) will be aborted.

12.5.1.7 File Processing

Due to the memory constraints imposed by the utilization of multiple I/O device drivers, LDR is designed to load most efficiently from the disk I/O device. When loading is requested from a file resident on a magnetic tape device, the file is first copied to disk. All subsequent accesses for this file will hence be made to disk.

In the normal loading process, LDR reads into its PP memory as many PRUs of input as will fit into the input area. For the disk file, three PRUs can be read at a time. A specialized loading scheme is utilized for the loading of absolute overlays in user call mode. This absolute overlay loader is described in greater detail in the next section.

As per system convention all programs within files are assumed to be separated by end-of-records.

A selective file search option is provided in the user call mode. When such an option is employed, files are searched and left positioned at the original starting location. More than one program or overlay with the same name or level numbers may exist on the same file. In such a case, the position of the program or overlay on the file is a contributing factor to the uniqueness of its identity.

12.5.1.7 Absolute Overlay Loading

Programs in absolute overlay format have been preprocessed by the GPSL and no longer require relocation and linking. Such programs are preceded by a one word (CM) header specifying the overlay level, the address at which it is to be loaded within the user's job area, and the overlay entry point.

In control card mode, the loading of programs in absolute overlay format can be detected only after the overlay header has been processed. In such a case, the 2LA overlay is loaded using the normal input scheme described in the preceding section. Once an absolute overlay is encountered on a file, all subsequent programs on the file must be in absolute overlay format to be loaded by this LDR access. This is necessary due to the retention of the 2LA overlay in PPU memory during overlay loading.

When processing in the user call mode, LDR can detect, from user call parameters, that absolute overlay loading is required. In this case, a special routine within LDR is utilized to load the absolute overlay. With the exception of the overhead required to process the overlay header during the first disk revolution, this routine will load absolute overlays at disk speed.

12.5.1.8 User Call Processing

"User calls" are calls from LOADER during the normal loading process or from OVERLOD during absolute overlay processing.

OVERLOD, a specialized version of LOADER which is assembled with the (0,0) level overlay, passes user call parameters directly to LDR. After performing the requested loading operation, user reply parameters (error flags, if present, and entry point address) are formatted directly by LDR into the user reply.

User calls to LOADER in the normal processing mode are pre-edited prior to calling LDR. In addition to the user call options specified in Section 12.1, LOADER formats its own call to LDR to search for and load unsatisfied external references from the library. User reply parameters to LOADER are formatted in the call constructed by LOADER for return to the user. User call formats are discussed in greater detail in Section 12.4.

12.5.2 Detailed Description

The following are detailed considerations for LDR. These considerations are meant only to amplify the flowcharts. LDR and its three overlay programs (2LA, 2LB, and 2LE) are described in Sections 12.5.2.1, 12.5.2.2, and 12.5.2.3, and 12.5.2.4 respectively.

12.5.2.1 LDR

LDR is composed of a series of modules which perform logical processing functions. These modules are defined and described in detail below.

1) INIT - INIT is the entry processor for the LDR program. The PPU input register is tested for the presence of "LDR", classifying the call as a LOADER or "user call". Absence of "LDR" indicates the call came from LOD or is a Control card call.

   The GPSL communication area from RA + 65, 66, and 67 of the user's job area is read and formatted internally. For a control card call, INIT tests bits 24-26 of RA + 67 to determine where to pick up the file name from which to load. If a "program call," the file name is found in RA + 64; if a "LOAD" control card, the file name is found in RA + 2. The FNT & library are searched respectively and the file read routine initialized if the file or program is found. If the name is not found, overlay 2LE is called for errors processing, an error message is output to the DAYFILE, and the job is aborted. A successful search will result in the file being read and a transfer to the CNIDENT module to begin table processing.

   If the call is from LOADER, user call parameters are formatted internally and status words are constructed for subsequent program use. The call types, as determined by the various parameter fields, are as follows: (See Reference Manual p. 3-41 ff. for parameter symbol definitions)

Call Type	Parameter Status	INIT Exit
Overlay Load	V≠ 0	OVLCAL
Unsatisfied External	UE(bit 42, word 2)≠ 0	UEXCAL
Overlay Generation	FN= 0	OVGEN
Total File Load	FN≠ 0, SL= 0, V= 0, UE= 0	FNCAL
Selective File Load	FN≠ 0, SL≠ 0, V= 0, UE= 0	FNSLCAL

OVGEN is a minor routine which fetches the file name from which overlays are being generated (from an internal LOADER table) and exits to the basic control card load entry. The remaining INIT exits are major rouintes and are described below.

2) OVLCAL - If the UE bit is set, the routine is searched for and loaded from the system library. Otherwise, the FNT only is searched. All searching is done against the FN field. If the overlay is resident on magnetic tape, transfer will also be effected to the 2LA overlay processor in order that the file can be copied to disk.

   Library and FNT disk programs are loaded using the special absolute overlay loader (loading at disk speed). FNT files will be searched end-around for the requested overlay.

3) UEXCAL - The programs in the SL list are utilized to search for matching entries in the library. Programs which have been previously loaded will not be reloaded. Exit is effected when all SL list entries have been processed.

4) FNSLCAL - The file specified by the FN entry is searched against the SL list. Programs previously loaded will not be reloaded. Searching continues end-around until all entries have been satisfied or the original file position is reached.

5) CNIDENT - CNIDENT is always entered when a table is to be initially processed. The table code number (CN) in the identification word is compared against a list of valid CN identifiers. A valid comparison will result in a vector jump to the proper table processor. If the absolute overlay mode flag is set, relocatable subroutine format tables will not be allowed Exit vectors from CNIDENT are summarized by the following table:

CN	Table Processor	Exit
50	Absolute overlay	Overlay 2LA
46	XFER	XFER
44	LINK	LINK
43	REPL	REPL
42	FILL	FILL
40	TEXT	TEXT
36	ENTR	ENTR
34	PIDL	PIDL
77	Ignored	ADVBUF
None	Edit for Loader Directive	LDTST

6) LDTST - Called when valid CN identifier cannot be found for table. Overlay 2LA is called to edit the field for a valid loader directive. Return from 2LA will be to ADVBUF to progress to the next relocatable subroutine format table (if the unidentified table proves to be a valid loader directive.)

7) XFER,LLINK, REPL, FILL, ENTR - Table processors which format named tables as entries in CM LOADER tables. Exit is always to ADVBUF to fetch the next table.

8) TEXT - Table processor which relocates program text to program area of the user's job area. Addresses are relocated as indicated by table relocation bits.

9) PIDL - Table processor to initiate processing for a program, includin, the initiation of a new table subgroup within the CM LOADER tables. COMMON allocation, if any, are saved in the LOADER tables and assigned storage addresses.

10) ADVBUF - The several table processors process a single CM word (5 PPU bytes) at a time. Each time a new word is required, ADVBUF is accessed to advance the buffer pointer to the next CM word. ADVBUF keeps track of the number of words to be processed for each table. When a table's word count has been exhausted, ADVBUF will, instead of returning to the table processor, exit to CNIDENT to classify the next table.

As all table processors access ADVBUF to advance the buffer pointer, this routine also provides the interface between table processors and the I/O routines. When all data has been processed within a buffer, ADVBUF acquires a new buffer before returning to the table processor of CNIDENT. Should an EOR or EOF occur, ADVBUF will honor the flag when applicable and execute the exit condition for the particular call being processed.

11) READ, READCM, READISK, READIT, COPYFIL - All normal processing (triple buffer) I/O interfaces are performed through the READ subroutine. FNT and RSLCLD table searches initially set up pointers to the table entries containing required I/O device and positioning information. When READ is called, it determines whether the required program or file is in CM or I/O device resident. If in CM, READCM is called to perform the physical loading. Otherwise, the file or program is on an I/O device and READISK is called. The input is read by means of READP request to the stack processor. READIT is called to do the actual formatting and issuing of the request. If READISK initially determines that the file is on a non-allocatable device such as magnetic tape, COPYFIL is called to copy the file to an allocatable device. The FNT is changed so that the file copied to the allocatable device then takes the name of the original file, and the original entry is discarded.

12) FNT, RSLCLD - Subroutines to search the system directories for matches against specified names. If a match is found, I/O interface pointers to the directory entry are set up prior to exit.

13) TSTCALR - Subroutine to store flag denoting error condition discovered by LDR or 2LA and call 2LE overlay to process error.

14) EXIT - EXIT performs a multitude of status restoration and initialization functions required prior to the exit from LDR and the activation of the CM LOADER. The clock time and LOADER table sentinels are updated; status flags and current CM core pointers are formatted and rewritten to the GPSL communication area in RA + 65, 66, 67; the FST entry is updated to reflect the next sector to be processed from the current file; activity on the control point is requested and the PPU dropped.

12.5.2.2 2LA Overlay

When an absolute overlay is to be loaded during control card mode, this routine is loaded into PPU core (starting at location 1773_8). Entry is made at 2002_8.

There are four types of loading that may occur. These are as follows:

1) When a LOAD card has been processed, loading will take place until an end-of-file mark is reached. When the file has been completely loaded, the PPU is released, and control is passed to the PPU idle loop.

2) When a program call control card is processed, loading will take place until an end-of-record mark is reached. When loading has been completed, the CP is activated, the PPU is dropped, and control is passed to the PPU idle loop.

3) When a user library call requests that an unsatisfied external be satisfied, the file is prepositioned. One record is then loaded. When the loading of an 0,0 overlay has been completed, the CP activated, the PPU is dropped, and control is passed to the PPU idle loop. If the overlay was not at the 0,0 level, instead of the CP being activated, return is made to the user (OVERLOD).

4) When a user requests an overlay to be loaded, a search is made for the requested overlay. The search is in end-around fashion, i.e., search to the EOF, rewind the file, and search to the starting position. When the requested overlay is found, it is loaded. If the overlay is not found, error number 74 is set up for error processing. After the overlay has been loaded, processing continues, as is defined in (3), above.

All of these loading types use the same routine to do the actual loading of the program into the user's job area. This routine (ABOVWR) determines where the program is to be loaded, and determines when the loading is completed.

When a loader directive is to be edited, 2LA is loaded into PPU core (starting at location 1773_8). Entry is made at 2000_8. The routines which edit the loader directives, and a brief description of each, are as follows:

1) LDPROC- This subroutine determines whether the loader directive is of proper format. Error numbers produced are 46, 47, and 50. (An explanation of the error numbers and the routine that handles them may be found in 12.5.2.4). If the loader directive is found to be in the correct format, one of the following subroutines is entered: SEGZERO, SECTION, SEGMENT, OVERLAY.

2) SECTION - Several conditions are tested for:

   a) To ensure that no segment or overlay cards have been previously processed--error number 51

   b) To ensure that no text has been previously processed--error number 71. The segment tables produced for a segment start at RA + 100 for that segment. If text has been processed, and the first segment is being processed, then the next available core location will be greater than RA + 100.

12.5.2.2 ZLA Overlay

When an absolute overlay is to be loaded during control card mode, this routine is loaded into PPU core (starting at location 1773_8). Entry is made at 2002_8.

There are four types of loading that may occur. These are as follows:

1) When a LOAD card has been processed, loading will take place until an end-of-file mark is reached. When the file has been completely loaded, the PPU is released, and control is passed to the PPU idle loop.

2) When a program call control card is processed, loading will take place until an end-of-record mark is reached. When loading has been completed, the CP is activated, the PPU is dropped, and control is passed to the PPU idle loop.

3) When a user library call requests that an unsatisfied external be satisfied, the file is prepositioned. One record is then loaded. When the loading of an 0,0 overlay has been completed, the CP activated, the PPU is dropped, and control is passed to the PPU idle loop. If the overlay was not at the 0,0 level, instead of the CP being activated, return is made to the user (OVERLOD).

4) When a user requests an overlay to be loaded, a search is made for the requested overlay. The search is in end-around fashion, i.e., search to the EOF, rewind the file, and search to the starting position. When the requested overlay is found, it is loaded. If the overlay is not found, error number 74 is set up for error processing. After the overlay has been loaded, processing continues, as is defined in (3), above.

All of these loading types use the same routine to do the actual loading of the program into the user's job area. This routine (ABOVWR) determines where the program is to be loaded, and determines when the loading is completed.

When a loader directive is to be edited, ZLA is loaded into PPU core (starting at location 1773_8). Entry is made at 2000_8. The routines which edit the loader directives, and a brief description of each, are as follows:

1) LDPROC- This subroutine determines whether the loader directive is of proper format. Error numbers produced are 46, 47, and 50. (An explanation of the error numbers and the routine that handles them may be found in 12.5.2.4). If the loader directive is found to be in the correct format, one of the following subroutines is entered: SEGZERO, SECTION, SEGMENT, OVERLAY.

2) SECTION - Several conditions are tested for:

a) To ensure that no segment or overlay cards have been previously processed--error number 51

b) To ensure that no text has been previously processed--error number 71. The segment tables produced for a segment start at RA + 100 for that segment. If text has been processed, and the first segment is being processed, then the next available core location will be greater than RA + 100.

For a detailed description of this overlay refer to section 12.5.2.1.

12.5.2.4 2LE Overlay

This routine is loaded into PP core (which contains LDR), starting at location 1173_8, when any error is detected by LDR or 2LA. The routine is entered at location 2000_8.

Essentially, the routine produces an error message and sets either a non-fatal or a fatal error flag. This flag may then be tested by the user when control is returned to the user.

Upon entry to 2LE, TEMP5 contains an error-message number. This number determines which error message will be sent to the dayfile. Message numbers $1\text{-}37_8$ are non-fatal errors. Fatal error message numbers are $41\text{-}77_8$.

The error message in the dayfile is preceded by a line which indicates that the error message was detected by the loader and which of the error flags have been set.

If the error message has been detected while processing an image, the image is placed in the dayfile immediately following the error message.

The messages are placed in numeric order. Error messages 1 and 41 are identical; hence to conserve space, essentially the first error message is 41. If any non-fatal error messages are added, a slight modification of the existing program will need to be made. The following is a list which contains an error number, the message produced, and the routine which detected the error.

ER1,(ER41): XXXX ERROR, CANNOT FIND FILE NAME - where XXXX is either "USER" or "CARD".

If LDR was called by LOADER a user call is being processed, hence "USER" is placed in the XXXX field; otherwise "CARD" is placed in the field.

ER1 is produced when the user has not placed a file name in the required parameter.

ER41 is produced when a file, which has been referenced in a call, cannot be found. The call may be either a user or system call.

Both errors will produce the call image which contains the illegal parameter.

Both errors are detected by LDR.

ER42: XXXX ERROR, FIELD LENGTH TOO SMALL - where the XXXX field is defined in the same manner as it is for ER1.

ER42 is detected by 2LA and LDR.

The ER42 error message is produced when the field length for the user's program is too small (the storage available between the starting point in the user area and the highest available location below the GPSL produced tables will not accomodate the user's program).

When this error is detected, the image being processed will not be placed in the dayfile. (The image being processed is most likely a binary card, hence the display code representation of it would be meaningless.)

ER43: BAD TEXT -

This error message is produced when LDR determines that an illegal TEXT Table entry is being processed; specifically, the relocation code is illegal.

When this error is detected, no image is produced following the error message.

ER44: FILE INITIALLY POSITIONED WRONG -

This error message is produced when LDR determines that an input file is initially positioned at an end-of-file mark.

When this error is detected, the image being processed (EOF) will not be sent to the dayfile.

The remaining error messages will cause the image being processed to be sent to the dayfile.

ER45: FIELD GREATER THAN 80 CHARACTERS -

LDR detects this error when a loader directive is improperly implemented.

ER46: ONLY ONE PARAMETER -

This error message is produced when 2LA finds an overlay loader directive with only one level-parameter.

ER47: INVALID CARD FORMAT -

2LA produces this error comment when it finds a termination character, "." or ")", prematurely implemented on a loader directive.

ER50: INVALID LOADER DIRECTIVE -

This error message is produced when 2LA determines that the first 7 characters on a loader directive card do not match one of the following 7 character words:

SEGZERO
SECTION
SEGMENT
OVERLAY

ER51: SEG OR OVERLAY CARD PREV PROCESSED -

A SECTION card cannot be used in the overlay mode. When it is used in the segmentation mode it must precede all segment cards.

When 2LA determines that the rules in the above paragraph have not been followed, the message for ER51 is sent to the dayfile.

ER52: SEGZERO HAS NOT BEEN PROCESSED -

2LA produces this error message in two instances; when a SEGMENT card is being processed and the required initial (SEGZERO) segment card has not yet been processed,and, when a SEGZERO card is being processed and an OVERLAY card has been processed. (Segmentation and overlay modes may not be mixed.)

ER53: SEGZERO SEGMENT NAMES DIFFER -

When there are too many parameters to fit on one SEGZERO loader directive card an additional SEGZERO card may be used. This card must define the segment with the same name. When the segment names differ for contiguous SEGZERO cards, Error Message 53 is sent to the dayfile. This is determined by 2LA.

ER54: NAME GREATER THAN 7 CHARACTERS -

When 2LA determines that a name used on a loader directive card is greater than 7 characters in length, this error message will be produced.

ER55: NO TERMINATOR FOUND -

When 2LA determines that a loader directive card does not have a legal terminator, "." or ")", this error message is sent to the dayfile.

ER56: INVALID CHARACTER -

This message is produced when 2LA finds a character on a loader directive card that is illegal. The legal characters are letters, numbers, parentheses, blanks, commas, and the period.

ER57: SEGMENT OR SECTION CARD PROCESSED -

When 2LA is processing in the overlay mode and determines that a SEGMENT or SECTION card has been processed, this error message is produced.

ER60: 1ST OVERLAY CARD HAS NO FILE NAME -

When 2LA determines that the first character of this first parameter on the initial overlay card is not alphabetic, this error comment is produced.

ER61: 1ST OVERLAY CARD LACKS 0,0 -

When the option to load the overlay a designated number of words above blank COMMON is used on a level zero overlay card, this error message is produced. 2LA detects this error.

ER63: 1ST PARAMETER MAY NOT EQUAL ZERO -

The zero level overlay may not have secondary overlay levels, e.g., 0,1 is illegal. 2LA detects this error.

ER64: ONLY 1 OVERLAY DESIGNATOR USED -

When 2LA determines that the user has not designated both a primary and secondary level on an OVERLAY card, this error message is produced.

ER65: C OPTION NOT LAST PARAMETER -

When 2LA determines that a termination character does not follow the C option on an OVERLAY card, this error message is produced.

ER66: TOO MANY CHARACTERS IN PARAMETER -

When 2LA determines that a level number on an OVERLAY card is greater than 77_8, this error message is produced.

ER67: C OPTION DOES NOT START WITH C -

The option used on an overlay card which allows the user to designate how many words above blank COMMON the overlay should be loaded. Must have the alphabetic character "C" as the first character.

When 2LA determines that the 1st character is not "C" this error message is produced.

ER70: DIGIT IS NOT OCTAL -

When 2LA determines that a digit used on an OVERLAY card is not octal, this error message is produced.

ER71: TEXT HAS BEEN PROCESSED -

When 2LA is processing a SECTION card and determines that text has been previously processed, this error message will be produced.

ER73: ERROR IN ABS. OVERLAY FILE FORMAT -

This error message is produced by LDR in two places; when LDR is in overlay mode and determines that the identification code of the subroutine being processed is not equal to 50_8.

ER74: REQUESTED OVERLAY PROG. NOT FOUND -

This error message is produced by LDR when absolute overlays are being input from a file.

It is also produced by 2LA when absolute overlays are being loaded from a file.

In both instances the file is searched in an end-around fashion, i.e., it is searched up to the EOF mark, the file is rewound, and the file is then searched up to where the searching began.

When the overlay program requested for input cannot be found on the file being searched, the message is produced.

12.6 LOADER, BOOKKEEPING AND PROGRAM LINKING/DELINKING

12.6.1 General

LOADER is the CPU portion of the GPSL which accomplishes the linking and de-linking of inter-program references, the processing of FILL and REPL (replication) tables and user calls, and the initiation of job processes.

The size and location of the LOADER varies with the type of processing being performed. Normally, the LOADER will be resident in the extreme upper end of the user's field length and consists of the following general format:

Communication areas	Loaded Programs	Unused core	LOADER tables	LOADER program
RA	RA+77			RA+FL

A special loader program called OVERLOD (OVERLOG if debugging routines are being used) is included in the (0,0) overlay for overlay runs. This routine elimates the need to retain LOADER in core while making user calls to load overlays at execution time.

LOADER consists of the following main routines:

1) CALL	(user call process)
2) CONTROL	(control card processing)
3) OVERLAY	(overlay generation)
4) OVERLOD, OVERLOG	(overlay loading - not integral with LOADER)
5) SEGMENT	(segment loading)
6) LINK	(for linking all external references)
7) CONTINU	(for normal Program Loading)
8) REPL	(for processing replication tables)
9) FILL	(for processing fill tables)
10) XFER	(for processing XFER tables)
11) CHKSUM	(for checksumming the remainder of loader)
12) MAP	(for building and writing out the MAP)
13) CLEANUP	(for performing functions for closing out a load)
14) SATISFZ (satisfy)	(for satisfying unsatisfied external references)
15) BLDCHK	(builds checksum for checking by CHKSUM)
16) USERSEG	(processes user calls for SEGMENT loads)

These routines will be explained in greater detail in Secion 12.4.2.

In addition to the above, LOADER contains a set of routines which are basic to the operation of almost all code in LOADER:

1) THREAD	(fetches the next entry from a given threaded list in the loader table structure)
2) SEARCHL	(retrieves the next external in the LOADER TABLES)
3) ENTSRCH	(searches for an entry table which matches a specific 7 character name)
4) RELOCAT	(performs necessary relocation of any legal address)

5) REQUEST (performs calls to LDR, LOD and MTR)

LOADER also contains a set of subsidiary routines which are usually used by one two or three of the main routines (1 thru 15) identified above:

1) SHUFFLE (restores parameters to their place at RA + 2 after processing of REPLICATION tables)
2) BLDINX (used by OVERLAY to compute indices for the processing of current overlays)
3) BUMPADD (used to update the addresses of instructions according to the string of data bytes in FILL and LINK tables)
4) FCHBYTE (used to retrieve one left adjusted 30 bit byte from the current thread being processed in the LOADER tables)
5) BLKOMN (allocates or reallocates the origin and length of blank common)
6) FCHCOMN (fetches the next common reference table from the LOADER tables)
7) FLUNSAT (fills in all unsatisfied externals with out-of-bounds references)
8) CONVERT (converts octal numbers of up to 30 bits to display code)
9) CONVADD (uses CONVERT to establish display code form of 18 bit address for core map)
10) BREAKUP (subdivides LOADER table entries into name--address for core map)
11) WRITE (sets up print buffer and accomplishes the output of core maps using WNX to write to the output file)
12) WNX (performs circular buffer I/O to output file)
13) TSTFILL (counts programs loaded from library by LDR in response to unsatisfied externals as well as user calls)
14) SEARCHU (utilizes SEARCH to find the next unsatisfied external in the LOADER tables)
15) PLUNK (places a specified address into the control byte of a given LINK table)
16) SCSGSRC (searches SECTION or SEGMENT tables for a given named entry)
17) LOADSEC (sets up the physical loading - by LDR - of all programs in a specified section)
18) LOADSEG (sets up the physical loading - by LDR - of all SECTIONS or programs in a specific SEGMENT)
19) PUTPROG (places a given program name in LDR parameter list for physical loading - calls LDR when the list is full)
20) CLRLIST (clears addresses in SECTION or SEGMENT list and counts programs actually loaded by LDR)
21) STOBYTE (places a 30 bit byte into the next available location in a thread in LOADER tables)
22) SQUEEZE (combines all LOADER tables into two condensed streams of LINK or ENTRY tables and moves the whole mess immediately adjacent to their associated segment)
23) USRFATLL (sets the fatal error list in a user call reply and skips over normal load processing)

24) HUNT	(searches for a SEGMENT of higher or equal level to the segment being loaded)
25) DELINK	(delinks all external references to levels of segmentation greater than or equal to the segment being loaded)
26) USERSCH	(searches for a matching entry when the K flag is on)
27) RESET	(establishes LOADER in original unloaded state for subsequent loading of "fresh" programs)
28) ENTFINE	(returns the address of a given entry point name, if such a name currently exists in the LOADER tables)
29) LOADPROG	(loads a program of a given name)

12.6.2 Detailed Description

12.6.2.1 Entry Points

LOADER entries from the PPU routines LOD or LDR are achieved by presetting the control point program address (word 0) prior to requesting the CPU. Entry points are described in the following table.

Entry Point	Description
LOADER	Normal entry from user calls. It sets up an exit from CHKSUM to go to CALL and then enters CHKSUM.
LOADER1	First entry to LOADER following a reload of LOADER into the user's job area. LOADER will call LOD if during the processing of a "user call" it has determined that the program LOADER is not intact. This entry is taken by LOD upon completion of the reload.
LOADER2	Normal entry to LOADER from LOD or LDR during control card processing when LOADER has been reloaded into CM.
LOADER3	Entry from LDR during control card processing when LOADER has not been reloaded.
LOADER4	Special entry made from LOD when overlays are being generated from input from more than one file.
LOADER5	Entry to LOADER from LDR when LDR has encountered a REPL table, but the replication is to be performed prior to the loading of more text.

12.6.2.2 LOADER Exits

The following exits may be made from LOADER:

Exit	Description
1.	After LOADER processes an EXECUTE, or Program call card, it transfers control to the job at the point specified.
2.	After LOADER completes processing a LOAD or NOGO it places END in RA + 2 and waits. This causes the CPU to be dropped and the system to advance to the next control card. (REQUEST)
3.	During the LOAD process LOADER calls LDR for I/O and library searching. (REQUEST)
4.	When LOADER discovers, by execution of the routine CHKSUM, that part of the LOADER program has been destroyed, LOD is called to reload the LOADER. (REQUEST)
5.	Any error which LOADER cannot handle will result in ABT being placed in RA + 1.
6.	During instant replication, when LOADER completes, it clears a flag (at FWALØDR-7), causing LDR to drop the CPU.

12.6.2.3 General Flow

Each of the individual routines are discussed in later sections. The purpose of the following is to demonstrate the operation of LOADER, the interaction of LOADER routines and the LOADER/LDR/LOD interfaces.

1) Standard Subprogram Loading.

A standard subprogram load is considered to be any program loading not involving segmentation of overlays. Most of the processing done for standard loads is also done for overlay generation and segment loading.

A standard subprogram load might consist of one or more relocatable binary programs, some of which require the loading of binary text from the system library. Assume that program decks A, B, and C (on file X) are to be loaded. As a further example, assume that B and C require programs SIN and ARCTAN respectively from the system library, and that ARCTAN requires programs SIN and SQROOT.

The loading process could be initiated by the control card sequence:

LOAD (X)

EXECUTE (Y)

where Y is an entry point in program A. The program loading, linking and execution would proceed in the following general manner.

When system program 2TS detects the LOAD control card it sets bit 23 of RA+67 to indicate a LOAD control card to LDR and LOADER. 2TS then calls LOD which, ascertaining that LOADER has not yet been loaded into the job area, does so. LOD then calls LDR which reads to CM the entire file X, storing all of the loader tables adjacent to LOADER and storing the binary text beginning at RA+100 or as directed by the text cards. During the storing operation, all addresses are relocated as required. When LDR has completed loading A, B, and C into CM, it requests activation of the CPU at that control point at the address in P which was preset by LOD.

Upon entry into LOADER at LOADER 2 the following events take place:

(a) The entire loader is checksummed and the result stored in CHECKSUM RA + FL-1. CONTROL is entered.

(b) CONTROL determines that LOAD control card processing is required and that there is no segmentation or overlay. CONTINU is then entered.

(c) CONTINU calls LINK for the linking of all references. Then REPL is called for processing or replication tables.

(d) CONTINU then calls REQUEST which places an END in RA+1.

System program 1AJ will advance to the next control card (the EXECUTE Y card) which 2TS detects is a LOADER card. 2TS calls LOD which checksums the CHKSUM routine and determines that LOADER is apparently intact. LOD then initiates LOADER at LOADER3.

Upon entry at LOADER3 the following occurs:

(a) The entire loader is checksummed and determined valid. Exit is made to CONTROL.

(b) CONTROL determines that it is processing on EXECUTE card and exits to REGEND which calls CLEANUP.

(c) CLEANUP calls SATISFY to fill in all unsatisfied externals within the program load being processed.

(d) SATISFY places the unsatisfied externals SIN and ARCTAN into a parameter list , and then calls LDR requesting the physical load of these routines.

(e) Upon completion of this loading by LDR SATISFY links up all references and processes any REPLICATION tables.

(f) SATISFY then loops back to find any further unsatisfied externals and discovers that ARCTAN has the unsatisfied external SQROOT.

(g) Steps d, e, and f (above) are repeated but on this subsequent pass no further unsatisfied externals are found and SATISFY then exits.

(h) CLEANUP then calls SHUFFLE to move any parameters (which have been saved within LOADER by LOD) to CM locations starting at RA+2.

(i) Next CLEANUP calls BLNKOMN to establish the origin of blank common at the <u>last word address</u> + 1 of the last program loaded (SQROOT).

(j) Once blank common has been established CLEANUP then calls FILL to process all fill tables (many of which will refer to blank common).

(k) CLEANUP next calls XFER to process the entry point "Y" which has been provided by 2TS from the execute card. The absolute address for "Y" is merged with that name and placed in location XFERTBL for later use by LOADER when it enters the loaded program.

(l) If the MAP bit - RA67 - bit 31 is a one and if the MAP (OFF) bit in RA+66 is not set, then CLEANUP calls MAP to produce the core map. The MAP bit in RA+67 will be on under normal control card loading such as this except when all programs loaded come from the library.

(m) Once mapping is completed, all unsatisfied externals are filled in with out of bounds references. This is left to last so that MAP can detect and printout any references which have not been filled in by LINK.

(n) Upon return from CLEANUP, REGEND sets the field length into AØ as required by Chippewa convention, and the first word address <u>for the next load</u> is set to the current value of the pointer <u>CORNEXT</u> which is the last word address of the current load.

(o) Next the RSS bit (which has been set in RA67 by LOD if DIS has set the corresponding bit for 1AJ) is tested. If it is on, LOADER terminates this phase of the job by placing END in RA+1.

(p) The transfer address is placed in B7, and entry is made to BLDCHK. The entire loader is then checksummed and the result stored in the last word of LOADER; BLKCHK then enters the loaded programs at the address preset in B7.

2) <u>Overlay Generation.</u>

The <u>control cards</u> for overlay generation and processing could be identical to those described under Standard Subprogram Loading. The major difference is that the first card on file X would be an OVERLAY directive in BCD format. In this case, when LDR detects the card it sets the Overlay Flag in RA+66 and builds a one word identifier in the LOADER table OVLINPT. This identifier contains the overlay levels L_1 and L_2, the first word address of the program

(FWA), and the entry point which is added later by LOADER.

Assume that file X is organized as follows:

OVERLAY 0,0	Prog A	Prog B	OVERLAY 1,0	Prog C	OVERLAY 2,0	Prog D	EOF 6 7 8 9

The first overlay encountered on the file must be Overlay 0,0 or the "MAIN" overlay. At least one overlay of level N, 0 must appear in the input file prior to any overlays of level N, M. (This is to permit establishing the proper starting location for each overlay.)

When LDR encounters the file X as above it will load the binary text for Prog A & B and the loader tables as described in the previous section. LDR then initiates the CPU at this control point at LOADER2.

Upon entry at LOADER2 the following occurs:

(a) The LOADER is checksummed by BLDCHK and the results stored away. Exit is to CONTROL.

(b) When Control finds that it is processing a LOAD card in overlay mode, it exits to OVERLAY.

(c) Upon <u>initial</u> entry, OVERLAY expects that the 0,0 overlay has been loaded starting at RA+101 with information stored in a table labeled OVLINPT.

(d) OVERLAY then attempts to satisfy externals through SATISFY. Then it processes FILL tables, RBPL, (replication table). and the XFER table. This completes the entity for overlay (0,0).

(e) Once the overlay (0,0) has been established any blank common declared is then origined at the LWA + 1 of that overlay, and CORNEXT is updated to reflect the space taken up by blank common.

(f) Since this is the first overlay and LDR has detected the fact that there are subsequent overlays the RELOAD bit will be zero. OVERLAY thereupon constructs a call to LDR to load the next overlay beginning at CORNEXT, with tables starting at TBLNEXT (<u>following</u> the loader tables for OVERLAY (0,0)).

(g) OVERLAY then attempts to LINK up all references within the newly loaded OVERLAY (0,0). Once this is completed, an attempt is made to link up any remaining unsatisfied externals in OVERLAY (0,0).

(h) Any further unsatisfied externals result in a search for corresponding library routines (using SATISFZ). Any

references to the entry point LOADER are replaced by OVERLOD (or OVERLOG) so that one of these routines will be loaded with the first overlay referencing LOADER. The XFER, FILL and replication tables are then processed for OVERLAY (1,0).

(i) Assuming that blank common had been allocated in the main overlay, all references to it in OVERLAY (1,0) are then linked up. Note that no externals in overlay (0,0) are linked to entries in 1,0 since they were filled in with out-of-bounds values prior to loading OVERLAY (1,0).

(j) On this pass through OVERLAY, the RELOAD bit will be a one, since LDR has detected that the next overlay is also at the primary level (2,0). This causes OVERLAY to output the overlay 1,0 to the last named file.

(k) Prior to initiating a CIO call for output, OVERLAY constructs a table header consisting of a CN of 50_8, L_1, L_2, the FWA of the overlay, and the entry point (EA) to that overlay.

(l) In addition a sixteen word area immediately preceding each overlay is established for a standard ID (77_8) table. The parameters for the CIO buffer are established such that output will commence at the FWA of the 77_8 table, proceed to the end of the buffer (16 words away) and then end-around to the beginning of the buffer, which starts at the (50_8) table header for the overlay.

(m) After I/O is complete, TBLNEXT and CORNEXT are reset to the values held prior to loading overlay (1,0). LDR is then called for the load of the next overlay.

(n) This subsequent load is treated as in steps g through l (above). However, prior to the initiation of the I/O of step 14,OVERLAY will detect that only overlay (2,0) must be written since bits will be set indicating that overlay (0,0) has already been written out to the file.

(o) Upon completion of the writing of the 2,0 overlay to the file, OVERLAY senses that on end-of-file has been encountered by LDR (the EØLØAD or end-of-load-flag = 1).

(p) OVERLAY then either exits to monitor or initiates loading of the overlay (0,0) by constructing a user call for that overlay.

3) Initial Segment Loading

Assume a load deck in file X of the format:

SEGZERO	Prog A	Prog B	Prog C	Prog D	Prog E	EOF

Where SEGZERO has the format SEGZERO (MAIN, Prog A, Prog B).

Further assume the following control cards are processed:

LOAD (X)

EXECUTE

When 2TS detects a loader related control card (LOAD), it calls LOD which loads the LOADER and calls LDR for the physical load of the file into CM. Instead of loading the program text and loader tables, LDR will load the SEGZERO tables beginning at RA+100. Since there are two programs plus a segment header, defined in SEGZERO, 3 words will be taken up by this segment table. The first word of labelled common or program text will be loaded later at RA+103. LDR then enters LOADER1 by initiating CPU activity at the control point and dropping the PPU.

Upon entering at LOADER1, the following occurs:

(a) The LOADER is checksummed by BLDCHK and the results stored away. Exit is to CONTROL.

(b) CONTROL determines that it is processing a LOAD card and that SEGMENT flag is set. Exit is to SEGMENT.

(c) SEGMENT is only entered as a result of a control card call. It uses the SEGZERO table to effect the loading of all programs for SEGZERO.

(d) Each name appearing in the SEGZERO table (other than the first parameter, the name of SEGZERO) is used to search first any other SEGMENT table, then any other SECTION tables.

(e) In this example, since there are no Segment or Section tables, SEGMENT will construct a call to LDR for the loading of Programs A and B, since they are defined as SEGZERO.

(f) Once these programs are loaded, they are linked up, and the Replication tables processed as in a normal load. SEGMENT then exits to monitor to initiate processing of the next control card.

(g) When the execute card is encountered, LOD enters at LOADER3, causing LOADER to checksum itself and transfer to CONTROL.

(h) CONTROL determines the existence of the EXECUTE card and exits to SEGEND to complete the load. Under these conditions SEGEND calls CLEANUP to fill out SEGZERO.

(i) Once all loading is complete, the loader tables for SEGZERO are moved down in core, immediately following SEGZERO, using SQUEEZE.

(j) Blank COMMON is origined immediately following the loader tables and all pointers set up.

(k) SEGEND then ititiates processing at the entry point specified in the last XFER card.

4) User Call Processing - Segment Mode

All user calls enter at LOADER which executes CHKSUM. If the CHKSUM test proves valid, control is transferred to CALL to process the user call. Assume a user call with the following key parameters:

FN = X

SL = pointer to a list containing the program names: D, B.

S = 1

$L_1 = 1$

The call would be processed in the following manner:

(a) CALL breaks down the user call and determines the legality of the call. Based on the contents of this call, CALL will exit to USERSEG.

(b) USERSEG calls HUNT, which searches through the segment tables to find a segment at a level equal to or greater than the segment level being requested in the user call, if one exists.

(c) HUNT establishes a pointer (FWADLNK) to the segment table for levels at or higher than the level to be loaded.

(d) USERSEG then calls DELINK to scan all segment tables below the pointer FWADLINK. During this scan all LINK tables which reference entry points at addresses greater than the

address in FWADLNK are "delinked". This process involves the clearing of the entry address in the link table to zero and the subtraction of this entry address from the proper location in each instruction for which a data byte exists. Out-of-bounds references are also thus delinked.

(e) Once delinking has been completed, USERSEG initiates the loading of programs from file X. In this case the SL list will contain program names D and B. Since there are no defined Segments or Sections, LDR will be called to load D and B.

(f) Once programs D and B have been loaded, USERSEG attempts to link up all references within the segment. Then all external references in SEGZERO are linked to the new segment. All external references in the new segment are then linked to SEGZERO.

(g) Once all references are linked, residual externals are satisfied from the library. The newly loaded library routines are linked within and between segments, as described above in Step (f). Then if there are any new residual externals arising from the library routines, they will be satisfied by repeating this Step (g).

(h) Blank common is then allocated (if this is the first declaration) and linked up. Fill and Replication tables are processed, the XFER table processed; and if the M flag is zero, a core map is produced.

(i) Unsatisfied externals are then filled in with out-of-bounds references if the F flag in the user call is a one.

(j) The loader tables for the new segment are then condensed and moved down adjacent to the LWA of the new segment by SQUEEZE.

(k) USERSEG then exits to SKPFLL1 which is a common routine for exiting from user call processing.

(l) SKPFLL1 provides the reply to the user call by clearing the first word to zero and forming the error reply bits, entry point 1 (EA) and entry point 2 (AA) into the second word of the call.

(m) Since there is only one call to LOADER (signified by a word of zeroes following the 2nd word of the parameter list), SKPFLL1 then returns to the instruction following the user call pointer to the parameter list.

5) User Call Processing - Overlay Mode

A special loader (called OVERLOD) is used for overlay loading. It is made part of Overlay 0,0 by LOADER during overlay generation. OVERLOD only checks the validity of the call and then calls LDR for the loading of the overlay. OVERLOG is used in place of OVERLOD if use of SNAP, TRACE, or labeled dumps is requested.

6) User Call Processing - Subroutine Loading

The same sequence is followed as for USERSEG with the exception that no delinking occurs, and the loader tables are not moved down adjacent to the last loaded program.

12.6.2.4 LOADER Routine Descriptions

The various routines which comprise the LOADER program are described below. All routines in LOADER call each other by the use of RJ X, where X is the entry point of the required routine (BLDCHK and CHKSUM are exceptions).

A standard scheme for register allocation has been followed throughout LOADER, with a few minor exceptions which are specifically identified by routine.

B1 = 1

B2 = On return from a subroutine B2 = 0 means that the subroutine did not accomplish what it was supposed to do, or that the end of a list has been reached.

B3, 4, 5, 6 = Transient

B7 = Normally contains the address of the PIDL table corresponding to the program being processed.

A1 = Fetch

A7 = Putaway

A0, 2, 3, 5, 6 = Transient

A4 = Address of current entry in a given list being processed.

XØ = Search keys

X1 = Fetch

X2 = Masks

X3 = Transient

X4 = Current entry in list

X5, X6 = Transient

X7 = Putaway

The following is a brief description of the operation of the major routines:

(1) OVERLOD

This routine is a small overlay loader which becomes part of OVERLAY (0,0). It validates the user call, and if valid transmits the interpreted parameters to LDR for loading of its requested overlay.

Validation includes:

(a) Parameter legality test.

(b) Relationship of call to OVERLAY mode, i.e., if a call is encountered for a normal program load, OVERLOD will request that LOADER be loaded in by LDR and then the user call is passed to LOADER.

During overlay generation all external references to LOADER are linked to OVERLOD; thus, entry to OVERLOD is the same as any LOADER call.

The following parameters in the user call are examined by OVERLOD:

(a) The overlay flag (V) - must be a one for overlay loading.

(b) Level numbers L2 and L1 must be valid.

(c) FN must contain a file name, and SL must be zero.

All other positions in the call are ignored. No mapping is done for OVERLAY loading.

(2) BLDCHK

This routine checksums the entire loader and places the result in RA+FL-1. The exit is taken by executing a JP B7.

(3) CHKSUM

This routine checksums the entire loader and compares the result with RA+FL-1. If RA+FL-1 ≠ 0 and the checksum agrees with it, the exit is taken by executing a JP B7. If the checksums do not agree, LOD is called to reload LOADER.

(4) CONTROL

This routine is entered either by LOD (when it discovers an EXECUTE or NOGO card) or by LDR as a result of a LOAD or program call card. By examining bits 24, 25, and 26 of RA+67 CONTROL determines the type of control card processing required. In addition, bits 19-20 in RA+66 indicate whether or not a SEGZERO or OVERLAY card were discovered by LDR during the loading. The following matrix gives the condition for entries into other LOADER subroutines.

CONTROL CARD	LOADER DIRECTIVE FLAG SETTING None	Segment	Overlay
LOAD	CONTINU	SEGMENT	OVERLAY
EXECUTE	REGEND	SEGEND	OVLEND
NOGO	REGNOGO	SEGNOGO	OVLNOGO
Program Call	PROGCAL	SEGCALL	OVLCALL

Note that SEGMENTS and OVERLAYS are mutually exclusive. The existence of both bits detected by CONTROL is prohibited by LDR. Therefore if CONTROL detects them both, the jobs will be aborted.

The entries CONTINU, REGEND, REGNOGO and PROCAL are all within CONTROL. The actions of each of these rouintes are as follows:

(5) CONTINU

This routine is called as a result of a LOAD card during normal loads. When it is entered all program text and tables have been loaded.

LDR will have loaded all loader tables with the PIDL table of the first program origined at FWALODR-4. CONTINU thus establishes the value of FWALNK (first word address for entry table search) as FWALODR-4. The limit for both entry and link table searches is set to zero, indicating that all tables in both strings are to be searched. CONTINU then calls LINK to link up externals and entry points in these two tables.

Using the same starting parameters, CONTINU then calls REPL to process replication tables.

Upon return from REPL, CONTINU exits to BLDCHK to rebuild the CHECKSUM and then places an "END" in RA+1 and loops awaiting the release of the CPU. All subsequent entries to CONTINU result in identical processing.

(6) REGEND

During normal loads, this is the last main LOADER routine to be entered prior to entering the user's program.

(a) If a DEBUG card has appeared (bits 28 or 29 of RA+66 set), ENTFIND is called to determine whether or not DEBUG has been loaded. If not, the routine LOADPROG is called to perform the load.

(b) The routine CLEANUP is called so as to complete the loading process.

(c) The field length is set in A0 so as to be easily accessible by the user. The current value of CORNEXT is saved in FWALOAD. This value of FWALOAD will be used in the next loader map produced for a user call during this run.

(d) If the RSS bit (bit 27 of RA+67) is set, LOADER goes into Recall until the bit is cleared.

(e) From this point on, the flow in REGEND is also used prior to user entry, for a segment load from control card. If NOGOFLG is set, it means that a NOGO card has been encountered. Exit is therefore made to ENDIT, so as to exit by way of END in RA+1.

(f) The transfer address is picked up from XFERTBL (FWALODR-1). If no transfer address was specified, this value will be equal to zero, and an error will result.

(g) The check is now made to determine whether this run involves TRACE or SNAP. If so, the debugging routines must be entered prior to entering the user. The address of SETADR (within DEBUG) is determined by calling ENTFIND. Finally, BLDCHK is called to checksum LOADER. The user entry address is passed on to SETADR.

(h) If TRACE and SNAP are not being used, it is still necessary to check whether or not a DEBUG card has appeared in the job, because if so, LOADER must make sure the DEBUG file has been updated to reflect the most recent load. Therefore, if bits 28 or 29 of RA+66 are set, ENTFIND is called to supply the address of WRDEBUG, an entry point in the routine DEBUG. Exit is made to WRDEBUG, the DEBUG file is updated, and return is made to step (i) below.

(i) The final processing in REGEND involves determining whether or not field length reduction is to take place. This code (CKREDUC) is placed near the end of LOADER to allow greatest chance for reduction when blank common overlays LOADER. Reduction will only take place when bit 32 of RA+66 is set (as a result of a REDUCE card for this load). Reduction is performed by placing a three-word long routine at the end of loaded core (CORNEXT plus blank common length) and entering it. This routine calls MEM to perform the reduction and then enters the user. Reduction is not performed if blank common extends past the reduce routine in LOADER.

(7) <u>REGNOGO</u>

This routine is entered as a result of a NOGO card during normal loading. NOGOFLAG is set and exit is made to REGEND.

(8) <u>PROGCAL</u>

This routine is entered as a result of a program call card during normal loads. LINK and REPL are called in the same manner as in the routine CONTINU. Exit is then made to REGEND, unless the program call was for SNAP or TRACE.

If the call was for SNAP or TRACE, then the situation is different. SNAP or TRACE are not actually to be entered at this time, nor is the load even complete. Therefore, REGEND is not called. Instead, the routine TSCARD (in DEBUG) is entered. This routine builds a TRACE or SNAP table from the parameters on the card and returns to LOADER. LOADER then places END in RA1, so that the system will advance to the next control card.

(9) <u>CALL</u>

This routine is entered by a call:

```
RJ LOADER
VFD 60/PARAM
```

Upon entry CALL locates the beginning of the parameter list by fetching the return address of the call from location LOADER. The first pair of parameter words and the parameter pointer are stored within CALL. CALL then validates the parameter as follows:

(a) The overlay flag must be zero.

(b) If the segment flag (S) is one, L2 = 0.

(c) Bit 28 of RA+67 (the MAP flag) is set according to the inverse of the NOMAP flag in the user call.

(d) If SL=0, FN is the name of an entry point to be found in the library, or a file name in the FNT.

(e) FWA is less than LWA.

(f) LWA is less than TBLNEXT.

CALL then exits to one of two routines: USERSEG or USERLOD. USERSEG is an entry point in the SEGMENT subroutine. CALLOK is contained within CALL.

(10) CALLOK

This routine processes user calls for non-segmented loads. The only fields of interest in the user call are Fn, R, SL, K, F, C, LWA, and FWA. CALLOK primarily preprocesses the call before passing it on to REQUEST and, following completion of LDR loading, CALLOK completes the loading based on the user call.

Preprocessing consists of clearing all unnecessary bits in the user call to zero to ensure a clear interface with LDR. X1 is then loaded with an LDR call pointing to the two word user call stored within LOADER. REQUEST is called to initiate LDR. When LDR completes the required loading it returns control to the CPU at REQUEST which will return to CALLOK.

Upon this return, CALLOK calls LINK and REPL. Upon return from REPL, the C flag in the user call is examined; if C≠0, then CLEANUP is called. Otherwise, FILL is called since the common references must be filled but no library routine need be brought in. The F flag is then examined; if F≠0, all unsatisfied externals are filled in and the non-fatal error flag is set if any are encountered. Once the load is completed, the reply is formulated and the user call replaced in its original location in the user's program. The next location in the parameter list is examined. If it is zero, CALLOK returns, to the user's program. Otherwise, the next parameter call is processed as described previously.

Note that each parameter word pair is processed to completion (or as far as permitted by the C and F flags) before the next parameter is processed.

(11) FILL

This routine is entered by a return jump (RJ) with the beginning address of the tables to be processed in FWAALL. FILL processes all fill tables appearing between the start address and table limit.

Fill utilizes a routine called FCHBYTE to fetch single, left adjusted byte from the stream of FILL tables. Every control byte (bit 50=0) contains an address relocation value. This value is relocated using RELOCAT and the resulting address is saved in X0. Then BUMPADD is called to add the address found in X0 to each instruction for which a data byte exists in the stream. Upon return from BUMPADD the stream pointer is stationed at the next control byte or the table terminator (a word of zeroes).

(12) REQUEST

This routine is a short program used for calling LOD and LDR. REQUEST places the PPU call in RA+1, clears bit 29 of RA+67 and then waits for it to be set by LDR when loading is complete. Once the bit is set, REQUEST exits to the calling program. In some cases the return to the calling program may be made by one of the LOADER01 or LOADER02 entries which make their return via an entry to REQUEST.

(13) REPL

This routine processes all REPL tables, starting at the address of the first loader table and proceeding until all REPL tables have been processed. The REPL tables are processed directly as indicated by the fields appearing therein.

(14) CLEANUP

This routine is used to complete loading and any bookkeeping required for completion of loading. CLEANUP is entered with FWA and LWA of loader tables to be processed. CLEANUP first calls SATISFY to fill in external references. Upon return from SATISFY, CLEANUP scans the program name list. If SATISFY has loaded any program during this load, bits 0-17 of the program name word $\neq 0$. In this case, CLEANUP calls LINK and REPL, giving as FWA and LWA the starting and ending locations of theloader tables for the programs just loaded. If no programs were loaded by SATISFY, CLEANUP then calls FILL to process all fill tables for the programs just loaded. CLEANUP then expands its scope of table processing to include all current loader tables in the job area by setting FWA to the first table and LWA to its last table. CLEANUP then calls LINK to link up all remaining references. Upon return from LINK, CLEANUP searches for all unsatisfied references in all LINK tables if bit 29 of RA+67 $\neq 0$ (fill flag). All such unsatisfied references are filled in with 377777. Bits 0-17 of each LINK table control byte are also filled in. Once complete, CLEANUP returns to the calling routine.

(15) LINK

This routine links up external references appearing in LINK tables beginning at FWALNK and ending with LINKLIM (or table sentinel) with entry points appearing in ENTRY tables beginning with FWAENTR and ending with ENTRLIM (or table sentinel).

LINK initializes the thread index (see THREAD) and thread limit for link table searching. Then SEARCHU is called to scan the tables for unsatisfied externals. If none are found SEARCHU will return with B2=0. If an unsatisfied external is found, the name will appear in X4. This name is tested to see if it is "LOADER". If so, and if we are in OVERLAY mode, the name is changed to OVERLOD(or OVERLOX if any debugging aids are being used), and the reference replaced in the LINK stream. The effect of this is to create an unsatisfied external called OVERLOD, forcing this special routine to be loaded from the library when SATISFY is called later.

If we are not in overlay mode the actual address for LOADER is placed in the entry by PLUNK. Then BUMPADD is called to add this address into every instruction for which a data byte exists in that particular stream.

If the unsatisfied external is not found, LINK preserves the present position in the link tables of the search and initializes the entry table search, and then calls ENTSRCH.

If a match is found for the name appearing in XØ, ENTSRCH will appear in bits 0→17 of X4. This address is replaced in the link entry and the result stored in the table. The address is then added into the referencing instructions by BUMPADD, and the search for unsatisfied externals is then resumed.

If a match is not found the address is set to zero and BUMPADD is called anyway. This is no more than a convenient way to position the link stream at the proper control byte following the one just processed.

(16) XFER

This routine searches for entry points for the last loaded XFER table, the next to the last loaded XFER table, or the specified entry point from an EXECUTE card or the name given on a program call card, depending on the type and mode of loading being undertaken.

If XFER processing results from a user call, then only the last two XFER tables are processed. Each name, in turn, is searched for in the entry table stream. If a corresponding entry point is found, the associated address is placed in the transfer table. If an entry point cannot be found for the last loaded XFER, a warning message is produced and the non-fatal error flag is set.

If XFER processing results from a control card, the routine utilizes the entry name provided in an EXECUTE card or the program name on a program call card. These names will be found in RA+64. If no name exists or no match is found, XFER then uses the last loaded XFER table to provide the entry point to the program.

(17) MAP

The MAP routine provides the core map for a given load. It is essentially a straightforward exercise in printline construction. The first line of the core map contains the following:

"COREMAP"
"Clock Time" of last LDR Load
"NORMAL/OVERLAY/SEGMENT"
L1, L2 or blanks
"CONTROL" or address of user call
blanks or 2 word parameter list in octal
First Word Address of load
Last Word Address of load
First Word Address of Blank Common
Length of Blank Common

The second line (if this is a full map) contains the header information for the first line. It follows, rather than precedes, this first line in case the P (partial map) bit is on in the user call. In that case only the first line and no others would be printed out.

The third line contains the First Word Address of LOADER and the First Word Address of the loader table.

The fourth line contains the header for the printlines to follow, which are a list of the program names loaded, and by program, the list of labeled Common blocks associated with each program.

The next mass of printing consists of all entry points in the load, with a cross reference for each entry point giving the absolute address of the reference, and the program name in which it appears.

Finally, if any unsatisfied externals exist a list of these unsatisfied externals will be produced with a cross reference for each one giving the absolute location of the reference and the program name in which it appears. This is not printed if the short Map option has been selected by means of a MAP(PART) control card. Note that this differs from the partial Map option described above.

MAP utilizes a circular buffer I/O routine borrowed from the FORTRAN compiler with a few modifications. Each printline is only as long as needed for the information in that line, with a 60 bit zero word terminator. The I/O routine (WNX) places this line into a 129 word buffer and attempts to maintain a continuous output stream to the buffer, as CIO moves the printers.

Upon completion of the MAP, an end-of-record is written and MAP delays until CIO has completed.

(18) OVERLAY

This routine may be entered from CONTROL (in the case of a LOAD card), OVLEND (in the case of an EXECUTE card), OVLCALL (in the case of a Program Call card) and OVLNOGO (in case of a NOGO card). A discussion of OVERLAY generation in this Section describes the general flow of this routine. The following are detailed points necessary to the

understanding of actual OVERLAY generation.

(a) The communications between LDR and LOADER during overlay generation are through:

1. EOLOAD bit - (End of LOAD bit) in RA+67, bit 30, indicates that an end of file has been reached or an overlay has been encountered on the file whose level is less than or equal to one in core.

2. WRTBIT - (Write out overlap bit) in RA+66, bit 23, indicates that at least one overlay should be written out in the current file.

3. OVLINPT - (The overlay Input Table) which contains the following entries.

   OVLZZHD, OVLPRHD, OVLSCHD (Overlays zero-zero, primary and secondary headers)

   OVLZZFN, OVLPRFN, OVLSCFN (Overlays zero-zero, primary and secondary file names for output)

   LASTOVF - (Last file name used to write an overlay)

   OVLINFN - (Current source file for overlay generation)

   OVLZBCM - (Blank common pointer and length for a given level)

   OVLZLWA - (Last word address of last loaded overlay)

4. Contained in each level overlay file name word (OVLZZFN, OVLPRFN, OVLSCFN)in the low order 18 bits is the first word address of the loader tables for that overlay.

5. Each header contains the level numbers and FWA of the overlay.

6. The sign bit of each OVLZLWA entry (when a one) indicates that the overlay at that level has already been written out.

7. Bits 23 and 24 of RA+66 form the binary pointer to the overlay to be processed by loader (i.e. - 0,1,2).

8. Bits 25 and 26 of RA+66 form the binary pointer to the level overlay which will be replaced upon the next call to LDR by LOADER.

9. The physical structure of the main LDR-LOADER communications table is:

OVLZZHD	50_8	L1	L2	FWA OVLZZ	entry point	Main (0,0) Overlay Entry
	OVLZZFN				OVLZZ TBLNEXT	
OVL PRHD	50_8	L1	L2	FWA OVLPR	Entry Point	Primary (X,0) Overlay Entry
	OVLPRFN				OVL PR TBLNEXT	
OVL SC HD	50_8	L1	L2	FWA OVLSC	Entry Point	Secondary (X,Y) Overlay Entry
	OVLSCFN				OVLSC TBLNEXT	
	LASTOVF					
	OVLINFN					
OVLZBCM(0) →					FWA Blank Common	Main (0,0) Overlay Entry
OVLZLWA(0) →					LWA+1 of overlay	
OVLZBCM(1) →					FWA Blank Common	Primary (X,0) Overlay Entry
OVLZLWA(1) →					LWA of Overlay	
OVLZBCM(2) →					FWA blank Common	Secondary (X,Y) Overlay Entry
OVLZLWA(2) →					LWA of Overlay	

Overlay already written bit

It can be seen that by utilizing the two, two-bit pointers provided by LDR in RA+66, multiplied by two, each required entry in the above table can be accessed.

(b) When LDR encounters an OVERLAY card the parameters file name, level numbers and first word address of the overlay are placed in the table and the indexes in RA+66 are set to the current overlay level just loaded.

(c) During the physical loading of programs constituting the overlay, LDR continues loading until another OVERLAY card or an end of file is encountered. The level of the next OVERLAY dictates the setting of the two bit replacement index in RA+66. If the level of this next overlay is less than or equal to the level(main, primary or secondary) of the overlay currently being loaded, the WRTBIT and EOLOAD bit are set by LDR.

(d) LOADER deals with all levels of overlays in an identical pattern; however, prior to processing overlay levels 0,0 the loader table pointers and blank common are initialized to "first load" condition.

(e) When entered for the first time, LOADER enters OVERLAY which accomplishes all the initialization for the main (0,0) Overlay, which has already been loaded by LDR. Thereafter, and until all overlay generation is complete (for the present input file) the routine OVERLAY deals directly with LDR, requesting the load of the next level overlay, and with CIO requesting the output of an overlay.

(f) Each overlay is linked, and all FILL, replication and XFER tables are processed and external references satisfied. After all possible library programs have been loaded to satisfy externals any remaining ones are filled in with out of bounds references, thus inhibiting any further linkage for that overlay. This process then inhibits the "linking forward" of references from a lower level overlay, to a higher level overlay.

(g) Blank common is then established (if this is the first declaration of blank common) or linked up to the appropriate references with FILL tables.

(h) Once Loader has completed all of this standard bookkeeping, the actual process of overlay generation takes place. If the WRTBIT and EOLOAD bit are not set,LOADER establishes the next available location for loading text and tables, and these pointers are saved in the table for the next higher overlay. Loader then calls LDR to load in the next overlay. If the WRTBIT is set, LOADER scans the table of overlay headers, file names and common pointers to determine which is the overlay to be written out to the output file.

(i) On the first occasion of encountering the WRTBIT, for example, LOADER may have loaded and linked overlay (0,0), (1,0), and (1,2); the WRTBIT being set during the loading of overlay (1,2), since each overlay just loaded has been established, it is now possible to write out all three, prior to loading the 1,3 overlay. This permits the ordering of overlays on the output file, in the same order in which they were loaded from the input file.

(j) LOADER processes each Overlay table, searching first for the lowest level that must be written out. This is established by the rule th t the lowest possible level not already written

out will be written out. Once an overlay is written out, bit 59 of the blank common pointer word for that overlay is set to one. This bit will remain a one until the overlay is replaced by a new overlay.

(k) Each overlay is output individually, using CIO. Using our previous example of overlays (0,0), (1,0), and (1,1); the process would be accomplished as follows. The header table for overlay (0,0) is retrieved, and the entry point inserted from the already processed XFER table. This one word table is then placed at the FWA of overlay (0,0)(normally location 100_8). A sixteen word card image is cleared, beginning at the last word address of overlay (1,1) (in the next available location in core) and the PIDL name and a 77_8 table CN code placed at that address. The CIO buffer parameters are thus established as follows:

FIRST = FWA, overlay (0,0)
IN = Last word address plus one of overlay (0,0) ($RA+100_8$)
OUT = Last word address+1 of overlay (1,1)
LIMIT = Last word address of overlay (1,1)+16

(l) When CIO is called, the overlay will be put out in a circular fashion from the resulting buffer, starting with the 16 word 77_8 card, proceeding for 16 words to LIMIT, then end-around to $RA+100_8$ to the overlay table header, and continuing to IN at the end of overlay (0,0).

(m) If SNAP, TRACE, or labeled dumps are being used during the overlay run, it is necessary to keep a record of the loader tables of each overlay for later use. Therefore in this case, an additional record is written on the overlay file after each overlay. It consists of the loader tables for all of the programs in this and associated lower level overlays. Thus, after a (3,1) overlay, the tables for the (0,0), (3,0), and (3,1) overlays are written. These records are given an identification of 76 so that LDR may skip over them while searching for overlays. The CIO buffer pointers are set up as follows when writing out the loader tables:

FIRST = TBLNEXT for the appropriate overlay
IN = FWALODR+1 (So that FWALODR will be the last word written)
OUT = Same as FIRST
LIMIT = FWALODR+2 (IN+1)

(n) Once all overlays are written out, that are required, OVERLAY proceeds to call LDR for the load of the next overlay and processing cycles as stated above. If the END OF LOAD flag was on, however, OVERLAY is done, since all overlays will have been written out by the above procedure. In the case where only three overlays (0,0), (1,0), and (1,1) have been generated before the end of file is encountered, OVERLAY will output the overlays in the same manner as k thru l (above), since the existence of either a WRTBIT or EOLOAD bit will trigger the output procedure.

(o) The determination of whether to proceed following this output process is made on the basis of the EOLOAD condition. If it exists, OVERLAY exits to OVLDONE, which determines whether the loading was initiated as the result of a LOAD or program call card. In the first case, LOADER exits by placing an END in RA+1. In the second case, LOADER creates a psuedo user call to LDR for the loading (first file built) overlay(0,0). Once LDR commences this load, control is transferred by LDR to the entry point of that overlay.

(p) The logic just discussed, results in several peculiar conditions:

More than one OVERLAY (0,0) may be generated. Which one to be loaded by OVERLOD is determined by position on the file, and the position of the file at the time of calling for that overlay.

(q) Any overlay may call any other overlay; however if a higher level overlay calls a lower level overlay or one of equal level, the LOADER system assumes that the user call will be wiped out by such action, and thus control is not returned to the CP followi-- the user call but is instead returned to the entry point established for that OVERLAY.

(r) This permits the exchange of control between main overlays for various systems.

(19) SEGMENT

The general flow of segment processing was covered in the discussion in section 12.6.2.3(4). Pertinent details are covered below:

(a) The initial entry to SEGMENT is initiated by control card processing of a LOAD or Program Call Card. The segment mode is established by the existence of a SEGMENT or SECTION card immediately preceeding any other records on the file used for loading. All SECTION cards must precede any SEGMENT cards, and such defined SEGMENTs must be of that level segment. Should the list of programs or sections making up a segment (or the list of programs making up a section) exceed a single card, the overflow may be carried on the following card if the same section or segment name is used as the first parameter of that card.

(b) When SEGMENT has been entered, only the SECTION or SEGMENT cards will have been loaded beginning at $RA+100_8$. The SECTION or SEGMENT table thus formed consists of a segment or section name (in display code, with bit postion 59 changed to a one) followe by a list of names in display code (with bit position 59 0). Thus a SEGMENT or SECTION definition is terminated when a new entry is found which is negative, or zero (since the section tables are separated from the segment tables by a word of zeroes and the segment tables are terminated by a similar word of zeroes).

(c) If any section tables exist, they always begin at RA+100_8, and continue to the first word of zeroes. If any segment tables exist (and at least segzero must be there!) they are located at the address found in bit positions 36 thru 53 of RA+65. Obviously, if there are no section tables, this address would be RA+100_8.

(d) Prior to the loading of any segments, one preliminary step must be taken if SNAP or TRACE are to be used. If bits 30 or 31 of RA+66 are set (the result of a DEBUG(T) or DEBUG(S) card, it is expected that the last control cards to appear were SNAP and/or TRACE cards, and that SNAP and/or TRACE tables will be present. These tables must be saved at this time, however, because once loading of SEGZERO begins, they will be destroyed. Therefore, the routine TSSEGUP (within DEBUG) is called to increase the field length and move the SNAP and/or TRACE tables up into the added field length. TSSEGUP returns directly to LOADER at step (e) below.

(e) Using this address, SEGMENT then calls LOADSEG, which will load the predefined segment whose first entry may be found at the address in register A1.

(f) LOADSEG begins by taking the first name in the list (following the segment name) and searching the list of predefined sections using SECSRCH. If a section exists whose name matches that entry, SECSRCH will return with B2 non-zero and the address of the matching entry in A1. If this is the case, LOADSEG will then call LOADSEC to load the predefined section beginning with that entry.

(g) Since a predefined section can consist only of program names, LOADER now has on its hands a ready made parameter list for calling LDR. All that is left to do is to search to the bottom of that particular section's name list, save the entry following it, and replace that entry with a word of zeroes. We then have a legitimate appearing parameter list. LDR is then called for a selective file load from the current input file, with a pointer setup for the first name in the given section table.

(h) Upon return from LDR, two things must be accomplished, the list must be restored to its original condition so that it may be used again, and a determination made as to the state of the loading process. First LOADSEC restores the last entry +1 to its original condition, wiping out the word of zeroes that was used to terminate the parameter list. Then LOADSEC returns to LOADSEG.

(i) The routine CLRLIST is then called to clear out the 18 bit address which LDR sets in each entry (if the program is loaded), then counts the programs loaded versus the programs requested, and lastly sets LODADDR to the address of the first program loaded for that section. This last little bit of fluff is for the use of USERSEG and has no importance in our current discussion.

(j) If the name found in the SEGZERO list is not a predefined section name, it is placed in a separate list at the address PARAMTR. When all names have been processed, (or the PARAMTR list has become overcrowded) LOADSEG sets up a call to LDR for the physical

loading of all programs in that parameter list. The loading is accomplished by a routine called NOWLOAD, which after return from LDR calls TSTFILL to process entries in the parameter list in the same manner as did CLRLIST.

(k) Once LOADSEG has completed the above activities, it returns to SEGMENT, which then compares the count of the programs actually loaded with the ones called for. If all of the programs defined in SEGZERO are not present in core, an error message is produced and the job aborted. If all are present, they are linked up and their replication tables processed.

(l) After the loading of the user's programs in SEGZERO is completed, the debugging routines are loaded into SEGZERO as required. The prior presence of any DEBUG card (bits 28-31 of RA+66 non-zero) will cause DEBUG to be loaded. SNAP and/or TRACE are then loaded depending on the presence of a DEBUG(S) or a DEBUG(T) card, respectively.

(m) SEGMENT now makes one of two possible exits. If a program call card was being processed, exit is made to SEGEND. Otherwise, a LOAD card was being processed, and exit is made by placing an END in RA+1.

(20) SEGEND

Once an execute card or program call card is encountered, the segment loading is completed by SEGEND. This routine accomplishes all the normal processing through CLEANUP, that is attendant to normal loading. Once all standard bookkeeping is completed, however, SEGEND then calls SQUEEZE to condense the loader tables into low core.

Upon return from SQUEEZE, SEGEND takes the pointer in A7 and places it in bits 36-53 of RA+67 so that the head of the SEGZERO tables can be located by HUNT during user call processing.

SEGEND then exits to the loaded segment by entering a location in REGEND (location CONT), at which the procedure is identical for normal and segment runs.

(21) SQUEEZE

SQUEEZE accomplishes the condensation of loader tables, and moves them physically adjacent to the last word address of the segment just loaded. The process is as follows.

Only LINK and ENTRY tables are processed. All such tables for the current segment are scanned beginning at FWALODR-4 and proceeding to the end of the chain. The putaways begin in increasing order starting at LWA+1 of the associated segment. This process thus reverses the order in which the entries will appear in the tables.

The ENTRY tables are moved first, one word at a time. If the ENTRY table has already been processed by LINK or MAP, the data byte (containing the relocation information) will already have been transformed into an absolute address which will appear in 0-17 of the ENTRY name word. If this has not been accomplished, then SQUEEZE must call RELOCAT to make this transformation. In any event, only the entry name word and the absolute address are moved down, while the second word is discarded.

The moving of LINK tables is a bit more complicated. First the format of the LINK table is modified somewhat. When LINK tables are maintained in high core as standard LOADER tables, each control byte contains an external reference name and zeroes in bits 0-17 if unsatisfied, and an absolute address in bits 0-17 if satisfied. Each data byte associated with that control byte contains the relocation information necessary to access the instruction which is referencing the named external.

The scan of LINK tables proceeds downwards through memory, beginning with FWALODR-4 while the tables are stored in increasing order, following the ENTRY tables. Not only does this step reverse the order in which LINK tables are encountered, but also the order of each LINK table. It is therefore necessary to save the control byte until all data bytes for the LINK stream have been moved to their new location. As each set of data bytes is moved, the related control byte is replaced at the head of the group so that it will be encountered first during LINK as is customary for normal loading.

After the last data byte of the entire LINK stream has been moved, it may be necessary to fill in one 30 bit group before replacing the last control byte so that the total stream will begin on an even word boundary. This last byte is constructed with an address pointing to a vacant space in LOADER, so that during the subsequent LINK processing of these tables modification by BUMPADD due to the entry will not destroy vital information. The location chosen was PPPPPPP which is the first word of the core map print buffer which is unused during linking.

As each data byte is moved down, its relocation information is reduced to an absolute address, thus minimizing future overhead, during segment call processing, for relocation of the addresses.

During the moving of both ENTRY and LINK tables all table headers, intermediate pointers (see THREAD) and sentinels are removed. Thus, there will be one continuous stream of all programs in the segment while each individual program loses its identity within the SEGMENT.

After all link and entry tables have been moved, a zero word is placed immediately preceeding the first word of its LINK table. A pointer word is then established, with pointers to the entry and link tables and finally a false PIDL header is constructed for the entire segment. The result appears thus:

	59–48	48–36	36–24	24–12	12–0
WORD 1	POINTER TO ENTRY TABLE	POINTER TO LINK TABLE	0000	0000	0000
WORD 2	Segment Level number	TBLNEXT for this Segment	0000	0000	0000
WORD 3	0000	0000	0000	0000	0000
4	Link Sentinal				
'	Link Table				
'	Entry Sentinal				
'	Entry Table				
'	Zero Word				
'	Programs in Segment				

Upon completing the above, SQUEEZE returns to the calling program with A7 pointing to the header word for the segment.

(22) USERSEG

This routine performs all user call processing for segment loading. It is entered from CALL after the basic user call has been edited. The following are specific details of operation.

(a) Upon entry, USERSEG calls HUNT to find the level of segment to be replaced (if any).

(b) HUNT performs two basic tests - searching for a segment at a level greater than the one requested by the user call and the setting of pointers between segment tables to maintain the thread for linked and delinked segments. On the first entry to USERSEG the only segment present in core will be SEGZERO whose LINK and ENTRY tables will be SQUEEZED down adjacent to the segment. HUNT begins by picking up the segment table pointer in RA+67 and using that address to access the segment table header for SEGZERO.

(c) HUNT then finds that the level appearing in the segment header is less than the one requested (assuming a call for a level greater than zero). The absence of a pointer in the header to any other segments indicated that the incoming segment will be an addition and not a replacement to the current core load. Further, this implies that the next segment will be loading starting at the next core location following the segment table header just "peeked at". HUNT terminates with this address in FWADLNR (first word address for delinking).

(d) On subsequent entries the activities of HUNT are more complex. Let us say, for example, that the first level loaded after SEGZERO was at level 1 while the next one to be loaded will be at level 3. Before HUNT is called for this second segment, the condition of the segments is as follows:

	59 54	35	18 17	0
Segment table header	L1	TBLNEXT for Segment	Downhill pointer	Uphill pointer

Downhill Pointer = address of next lower level segment table header

Uphill Pointer = address of next higher level segment table header

(e) On this second scan HUNT first fetches the initial pointer from RA+67. The segment table thus accessed will be the SEGZERO table. HUNT determines that the level of the called overlay (L1=3) is greater than the one in the table (L1 0). Since a pointer exists to another higher level segment (uphill pointer≠0) HUNT uses this address to access the next segment table.

(f) When accessing this table HUNT detects that the downhill pointer is vacuous and sets it to the address of the SEGZERO segment table header, HUNT also determines that the level of this next segment is still less than the one being called and that there are no higher levels present (uphill pointer=0). FWADLNK will then be set to the next core location following the segment table header.

(g) Assuming that the next user call is for a segment at level L1=2 HUNT will search for a greater or equal level as shown above, but when the level 3 segment table header has been encountered, HUNT will set FWADLNK with what would have been the downhill pointer which points to the last encountered table header.

(h) Once HUNT has completed its search USERSEG sets up CORNEXT to the value FWADLNK + 1 and initializes loading pointers. If a total file load is required, a simple single LDR call is made.

(i) If an SL list exits USERSEG commences to process it in the following manner:

1. Each entry is fetched from the SL list and SEGSRCH (search predefined segment list) is called. If there is a defined segment name matching the name from the SL list (B2≠0) then LOADSEG is called to load the segment. (Action described under SEGMENT)

2. If there is no matching name, SECSRCH (Search for predefined Section) is called. If there is a defined section (B2≠0) USERSEG calls LOADSEC (action same as in SEGMENT) to load the section.

3. If the given SL entry results in its loading a predefined Section or Segment the CLRLST routines will have set LODADDR to the address of the last program loaded for that Section/Segment. Upon return from the loading process, LODADDR is placed into bits 0→17 of the particular SL entry.

4. During this pass over the SL lists all names not identified as SEGMENTs or SECTIONs are ignored.

5. When the full list has been processed only these entries which do not get processed (either single program names or misspelling) will have their address fields still zero. USERSEG then calls LSR with the full SL list as parameter; however, LDR will ignore all entries already filled in.

(j) Once all specified programs have been loaded, the values FILLOUT and INCOMPL are composed. If unequal, the number of programs loaded does not match the number requested and USERSEG puts out an error comment and the non-fatal error flag is set for the USER CALL reply.

(k) If the load was complete, USERSEG performs linking and REPL and XFER table processing. Any unsatisfied externals remaining after the linking has been completed will then be satisfied by calling SATISFZ.

(l) There are two exits from SATISFZ which may be used to return to USEREG. The first, normal exit is taken when no programs have been loaded from the library during that particular entry to SATISFZ. The second exit is set in B7 by the calling program. It is taken when at least one program has been loaded by LDR.

(m) If a program has been loaded from the library to satisfy an external it will most likely possess a number of externals some of which may be linked within the segment, others will be limited to lower level segments and some may again be satisfied from the library. For this reason, the exit address in B7 will return to the linking processor of USERSEG. The normal exit proceeds to the remaining portion of USERSEG.

(n) Linking of all programs loaded proceeds in the following manner:

1. FWAALL, FWAKLNK and FWAENTR are set to FWALODR-4 as in normal loading. When LINK is called, this effects the linking of all LINK and ENTRY tables for the programs just loaded which make up the new incoming segment.

2. FWALNK is then set to FWADLNK, and LINK is called again. This will link up externals in the next lower level segment (remember that FWADLNK is set to the address of the segment table header for the last segment to be saved) with entry points in the incoming segment.

3. The downhill pointer is then placed into FWALNK. (Aha! That's what that thing is for)and LINK is again called. This causes the linking of all external references in the segment immediately lower in level than the last one processed in step 2 above to the entries in the incoming segment.

4. This process continues until a downhill pointer of zero is encountered, indicating that there are no further lower level segments. Matters are then reversed with FWAALL being replaced in FWALNK, and FWAENTR is set to FWADLNK. When LINK is called, this time, any unsatisfied externals in the new segment will be linked to entry points in the next lower level segment.

5. Entry points in all lower level segments are linked to the current new segment by processing each set of segment tables until a zero downhill pointer is encountered. (In the same manner wherein the link tables for lower level segments were handled.)

(o) Once all linking of programs and library routines has been accom--plished, a coremap is produced if the M bit in its user call is zero. Then all unsatisfied externals are filled in with an out of bounds reference of 377777B if its F flag is a one.

(p) The loader tables are then condensed and moved down using the SQUEEZE subroutine (described in SEGMENT). When SQUEEZE is complete, the uphill pointer in the segment header, at which FWADLNK points, is updated to point at the new segment header just created. During the loading of each segment, the main loader tables from previous segments are discarded, since all needed information is kept in the squeezed tables. However, if the debugging aids are used, it is necessary to have record of all tables for all segments in core, so these tables are preserved. The correct tables are saved during each load by using the TBLNEXT values kept in bits 36-53 of word 2 of each set of squeezed tables.

(q) USERSEG, after some final bookkeeping, exits to the terminal processing in the user call processor CALLOK.

(r) LOADPROG

This routine issues a user call to LOADER in order to load a program of a given name. Since this means a call to LOADER is being issued within LOADER, great care must be taken. This routine cannot be used if LOADER is already processing a user call. It is used to load the debugging routines (SNAP, TRACE, or DEBUG) when it is necessary to do so, and this involves calls from REGEND, SEGMENT, and PROGCAL. Since the checksum must be verified as soon as LOADER is entered, LOADPROG calls BLDCHK just prior to issuing the RJ LOADER.

12.6.2.5 Special LOADER Routine Descriptions

There are several routines which are the heart and soul of LOADER, and whose logic should be understood if one is to perform intelligent maintenance of LOADER. These are THREAD, SEARCHL, ENTSRCH, RELOCAT and REQUEST. Their operation is detailed below:

1) THREAD

This routine accomplishes the fetch of a single entry from a specified table in the loader table area. It operates on a threaded list of loader tables produced by LDR during the physical loading process. The structure of these threaded tables can be found in section 12.2, INTERFACES. Referring to diagram above the access of a table proceeds as follows:

(a) Prior to entry to THREAD a 60 bit word called THDINDX (thread index) is set up . A MACRO has been established which does this, called SETHRD. The form of the macro is SETHRD address, N where address is the location of the first PIDL table for the first set of programs to be processed and N is a number from zero to four which gives the position (left to right) of the table pointer to be used in the loader tables. For example if N=0, then the first or "alfa" pointer would be used, indicating that LINK tables are to be accessed. (See diagram of tables).

(b) The result of executing SETHRD is to form an index word whose various fields are:

N	Address of Current PIDL	Address of Next Table	Address of Next ENTRY

(# of bits from right of word to THREAD Pointer)/2

(c) On first entry to THREAD, note that all but the address and N fields are zero, since these are the only ones established by SETHRD. THREAD is subdivided into three portions; Fetch the next word, start a new table in the same program string, and start a new string for a new program PIDL. On first entry it is necessary to do all three of these things. First, given the address of the PIDL header, THREAD must access the appropriate pointer, designated by N and with this pointer, and the PIDL address build the address called NEXT TABLE. This address is inserted in the THDINDX. Next THREAD processes the specified table header which may contain a pointer to the next table in that program chain. This address is placed in THDINDX. Upon completion of this process, the address of the first word of the given table (after the table header) is available in A4 and the first word is in X4. The pointer in A4 is also placed in THDINDX, thus filling in all fields. THREAD then exits with the required entry in X4.

(d) On a subsequent entry to THREAD, the THDINDX is fetched and the next word address taken taken from bits 0-17. This word is fetched into X4, and if bits 18-59 are zero, a sentinel (end of table) has been found. If not, THREAD exits normally with B2/0.

(e) If a sentinel has been encountered, then THREAD must determine if any other tables are in the chain. The next address in THDINX is then obtained through shifting, and if zero, it indicates that the end of that particular chain has been reached. If there is an address, the word at that address is fetched (it will be a new table header) and the NEXT TABLE pointer from that header will replace the old pointer in THDINDX. The next word (in descending order) will be the next entry to be returned by THREAD, which will exit normally after putting that address in the last field (bits 0-17) of THDINDX.

(f) If the address was zero, indicating that there were no more tables in the string, the next program PIDL table must be accessed to see if there are any more tables to be processed at all. This is accomplished by taking the address field of the current PIDL and fetching the pointer word at PIDL-1. The omega pointer (bits 0-17) gives the relative location of the next PIDL from the current PIDL. If it is zero, there are no more program tables to be processed, and B2 is set to zero and THREAD exits. Otherwise the address of the new PIDL table is replaced in THDINDX and then the process of getting a new table, and a new entry in that table are undergone, with THREAD exiting with the entry in X4 and B2≠0.

(g) Each time a new PIDL is accessed, the address of that PIDL is compared against THDLMIT (thread limit) and if it is less than THDLMIT (remember we are in descending order) the search is terminated, B2 is set to zero and THREAD exits.

We therefore have the concept of a string of loader tables beginning at FWALODR-4 which are grouped by program and headed by a PIDL header and pointer word. One may proceed from PIDL header to PIDL header down this chain using the "omega" pointer word. Using THREAD we can proceed down the chain of LINK, ENTRY, FILL, and REPL tables, within each program group, for each program in the chain. We will call the major chain of LINK tables, specified by the PIDL header pointer, the THREAD. Within the thread there are one or more tables, which do not correspond to the exact tables produced by the compiler and assembler. These tables are linked together by the pointers in each table header. This scheme permits the generation of any type of table at any time in the input stream, thereby eliminating the need for the compiler or assembler to gather together all LINK or ENTRY points for issuing tables to the source stream.

It is obvious that the current position in the thread may be saved by preserving only the thread index. Thus during the processing of LINK tables THREAD will be called upon to retrieve entries for ENTRY tables for each LINK entry, requiring that the thread index for LINK scanning be saved in LNKSAVE,aand the thread index for ENTRY processing be saved in ENTSAVE by the calling program.

2) SEARCHL

This routine fetches the next control byte (60 bits containing EXTERNAL name and either zero or an assigned address in bits 0-17), into X4, from the LINK thread, beginning with FWALNK and ending with LNKLMIT.

First the THDINDX is set up with FWALNK and the LINK pointer, and LNKSTRM=0. Then SEARCHL calls FCHBYTE to procure one 30 bit byte from the stream. If the most significant bit (bit 59) is a zero, and the byte itself is non-zero, we have found part of the control byte. This half is then saved and FCHBYTE called to retrieve the second half byte which is shifted end around and merged with the first byte. SEARCHL then exits with B2≠0 and the full 60 bit control byte in X4.

If a control byte cannot be found in the remainder of the thread being searched, FCHBYTE returns with B2=0, and SEARCHL then exits directly to the calling program.

The approach taken here is due to the fact that control bytes, although sixty bits long, may be split between computer words, to decrease the core storage requirements of the LINK stream, which can get quite long. The variable LNKSTRM is a flag indicating whether the left or right half word is to be accessed next. If the sign bit (59) is a one, the address contained in bits 0-17 of LNKSTRM points to the word which must be accessed to retrieve the right half byte. If the sign bit is a zero, the next word in the LINK stream is fetched using THREAD, and the left half byte isolated. For this reason, LNKSTRM is set to zero upon first entry to FCHBYTE from SEARCHL to ensure that the scan begins at the head of the stream.

3) ENTSRCH

This routine searches for an entry in the ENTRY table whose bits 18-59 match the corresponding bits in X0. ENTSRCH sets the THDINDX to FWAENTR and the THDLMIT to ENTRLIM (which is usually zero). Then an entry is taken from the

ENTRY table using THREAD. If thread returns with B2=0, no more entries are available in the table, and ENTSRCH exits immediately. If an entry is found it is compared with the key in X0. If a match is not found, ENTSRCH loops back to get another entry. If the two names match, ENTSRCH exits with B2≠0, the first word of the ENTRY pair in X4 and the address of that entry in A4. If this matching entry has not been previously accessed, bits 0-17 of the entry will be zero. In this case, the second word of the table (at A4-1) is fetched and the relocation information extracted. The routine RELOCAT is then called to provide an 18 bit absolute address which is derived from the base address and relocation bits. This address returned by RELOCAT is merged into bits 0-17 of the entry and this is replaced in the ENTRY stream. Subsequent accesses of this table will then not relocate this address again.

ENTSRCH thus returned to the calling program with the entry and an 18 bit address in X4.

4) RELOCAT

This routine computes an absolute address from a relative base address and relocation flags positioned as a 30 bit byte in the upper half (bits 30-59) of the X4 register. The base address has a quantity added to it whose character is determined by the relocation bits: An RL of 0 means no addition

An RL of 1 causes the PIDL header for the program tables being processed to be fetched. This header contains the the FWA of the program in bits 0-17. This address is then added to the base address to make an absolute address out of an address which is relative to program origin.

An RL of 2 causes the fetching of the PIDL header and this address is complemented and added to the base to provide an absolute address for one which was relative to the complement of program origin.

An RL of 3-77 causes the fetching of the corresponding entry following the blank common entry in the PIDL table. The address computation is the same as for blank common relocation.

The result of this computation will be found in bits 0-17 of X3 while the original entry will remain in X4.

12.6.2.6 LOADER ERROR MESSAGES, MEANINGS AND RECOMMENDED ACTIONS

1. OVERLAY CALL FROM RELOCATABLE

A relocatable program has made a user call for an overlay load. This is one of the few redundancy checks made to validate user calls. This mode of operation is illegal, since once overlay loading is initiated, control is taken away from the user call and returned to the called overlay. It is possible to accomplish the same activity, if the file named in the user call contains all absolute overlays. This will trigger absolute mode, and if Overlay(0,0) has not been called for, control will return to the user call.

2. FIELD LENGTH NOT SUFFICIENT FOR OVERLAY GENERATION

This is self explanatory. During OVERLAY generation room must be allocated from the main LOADER of about 3500 locations and all loader tables for a given core map. During OVERLAY execution a great deal less core is required.

3. CANNOT COMPLETE THIS OVERLAY, BAD INPUT

During overlay generation, if LDR encounters difficulty in loading text or other tables, it will produce an appropriate message such as "USER ERROR, BAD TEXT TABLE". When LOADER commences to complete the overlay the fatal error bit (set by LDR) is detected, and the above message put out and the job aborted. The core map will contain the last good overlay generated.

4. LOADER TABLES GARBAGE, OR OVERLAY SEQ ERROR

LOADER tables are threaded in a list below the LOADER in CM. If these tables are destroyed by a user program (in normal mode only), the THREAD routine will be unable to cope with the situation, the above message will be output, and the job aborted. If an attempt is made to generate a secondary overlay, without first having generated its corresponding primary overlay (or zero level) the THREAD routine will have the same difficulty, and this message will result.

5. WARNING BLANK COMMON GREATER THAT PREVIOUS DECL

If blank common has been previously established in a set of programs (Segment, Overlay or a file loaded as the result of a control card) all subsequent references must not be greater than that which has been allocated. This is a warning only and does not abort the job. However none of the references to blank common are truncated, so it is quite possible for a program to destroy itself, if the caution is not heeded.

6. BLANK COMMON EXCEEDS AVAILABLE CORE, TRUNCATED

During blank common allocation, no length will be established which would exceed FL or overlap the 20 word residence set aside for the LOADER. As in 5 (above) no reference is truncated, only the allocation, for core map purposes.

7. REPLICATION EXCEEDS AVAILABLE CORE

During the replication process each putaway is checked to see that it doesn't destroy the LOADER or its tables. If this should occur, the replication table processing is terminated but loading and execution are continued.

8. SOURCE IS ZERO IN REPL

If no address is furnished for the source stream to be fetched from, this warning is given. The source is then set to location RA+1 (which usually contains zero) and replication initiated.

9. LEVELS NOT PERMITTED IN STANDARD CALL

If the S and V bits are not on in the user call, it is interpreted as a "normal load". If L1 and L2 are non-zero, it is possible that the call was intended as an Overlay or Segment call but the appropriate bit is missing. This warning is put out, and processing continues as if a "normal" load.

10. SEGZERO INCOMPLETE, JOB ABORTED

If all of the programs specified on the SEGZERO card cannot be found on the input file, this error message results and the job is aborted. Please note the distinction between the necessity for finding all programs explicity called for,

and the possibility that not all external references from these programs can be satisfied. In the latter case no error comment is made, since this is a normal condition for a segmented job.

11. SEGMENT CALL, BUT NO SEGZERO PRESENT

This is the result of a user call for a segment load. If it is accomplished from a Normal or Overlay program, it is impossible to establish the necessary delinking point, since certain pointers are established for all segment loading during the initial loading of SEGZERO. This is probably the result of an erroneous setting of the S bit.

12. SYSTEM ERROR IN LOADER..HELP..CALL CDC

This is explanatory. LOADER has been designed to "FAIL SAFE", that is,all communications with the system are checked out and edited in some manner. If one of the interfaces degenerates or a new "bug" appears in LOADER, the communication checking can result in this error comment.

13. SEGMENT REQUEST WHILE IN OVERLAY

In general the loading of segments and overlays is mutually exclusive, although normal loading may be done by both. This indicates that OVERLOD has called for the loading of a segment. The present implementation cannot permit this, since the original pointers to segment tables will not have been preset by the SEGZERO processor.

14. SL LIST IS EMPTY, FATAL ERROR

This is the result of a user call containing an SL pointer which gives an address at which there isaa zero word (signifying a vacuous selective load). It would be unusual for a programmer to incur the overhead for such a call when there are cleaner ways, thus this condition is assumed to be an error, and a fatal error bit is returned in the user reply.

15. NO PROGRAMS FOUND FOR SEGMENT

This is the result of a user call where not one of the programs requested could be found. Since this would aid the programmer in tracking down an error (most likely a misspelled file name) it is included, although it performs the same function as the following message.

16. REQUESTED SEGMENT INCOMPLETE

This comment results from a user call and is produced for the same reasons as SEGZERO INCOMPLETE. Please note again that this message does not result from an inability to satisfy all externals, but instead arises when all the programs explicitly requested cannot be located.

17. UNSAT BIT NOT PERMITTED, EXCEPT IN OVERLAY MODE

There are two ways in which programs may be called from the library. A single program can be loaded by putting the name in FN and setting SL=0. If it is desired to load an absolute overlay from the library, LOADER must be informed by setting the overlay bit and the unsatisfied external bit. If this bit is set at any other time, it will override the error flags returned by LDR, and could cause LOADER to begin loading absolute overlays on top of proper programs. This condition is thus prohibited.

18. WARNING NO MATCHING ENTRY FOR XFER

During XFER table processing if a non-zero or non-blank entry point is found in the XFER table, all available entries in the LOADER table are searched. If one matching cannot be found, this warning is put out. If this was the result of an EXECUTE card or program call card the job is aborted; otherwise the non-fatal error bit is returned to the user.

19. CANNOT COMPLETE LOAD, JOB ABORTED

This is intended to correct fatal error conditions which occur in control card mode when LOADER requests LDR to load library routines. Since a request of this type appears to LDR as a user request, 2LE, after giving a fatal error message, returns to LOADER rather than aborting. Thus, an error message from 2LE appears before this message.

20. NO TRANSFER ADDRESS

If, at the completion of processing a load, no program provides a transfer address, the job is aborted with this message. This condition will not occur if the load is completed with a NOGO card or an EXECUTE card which specifies an entry point.

12.7 CP LOADER

12.7.1 Introduction

The purpose of the CP LOADER is to provide the same functions as the standard GPSL Loader ones but faster than GPSL.

The objective is to relocate the text in normal load mode, at the disk transmission in 6600.

It is expected to lose no more than one disk revolution out of ten by using 6638 and 6600 with a circular buffer size of 2000_8 CM words, and one disk revolution out of two with a 6400.

The improvement of the loading speed is due to the following reasons:

- PP programs are only used to initiate physical reading and to provide some CMR information.
- TEXT is relocated by a CP program by executing e very efficient stack LOOP.
- An External Reference Table has been implemented in the directory, so that it is possible to set up the list of all library routines which are necessary and to load them with one PP request only.
- Buffer for mapping has been increased, we use for that all the CM space which is available.
- To load CPLOADER needs only one disk access and the relocation of CPLOADER is performed by itself.
- A new high speed read function has been implemented in the stack processor.

With this new function, the stack processor does not stop the read function at EOR. Also it does not lose one disk revolution each time it has to recompute the buffer parameters. So that it is possible to read a file in its entirety at the transmission speed with a normal buffer length.

CPLOADER and standard GPSL can coexist in the system. A new control card: LOADER has been implemented in order to determine what LOADER is to be used: LOADER (PPLOADR) indicates standard GPSL, LOADER (CPLOADR) indicates CPLOADER; default option is an installation parameter:

```
IP.LDR = 0 PPLOADR
IP.LDR = 1 CPLOADR
```

The external specifications are exactly the same as the standard GPSL ones.

Except in search file mode, the file is searched twice starting from the last position and if the requested program is not found, the file is positioned at End of File.

In case of fatal error, the name of the program being loaded is indicated by a message.

The action of CP Loader is irrelevant when attempt is made to load a program which has been assembled in absolute mode and if this program is such that some information is written in RA+1 at the loading time (ORG 1 for instance).

About the internal structure, the part of CPLOADER which performs the normal loading is only present in CM permanently; four "relocatable overlays" of CP Loader have been provided to perform special functions: Segmentation, Overlay generation, Mapping, processing of fatal errors.

The structure of the LOADER tables is the same.

Some areas have been added to the Loader tables:

- circular buffer,
- the list of all programs which have been loaded,
- the list of all common blocks,
- an area to load LOADERV or LOADERS overlay.

These areas are put together in the same part of CM, between the CM loader tables and the first word address of Loader. A pointer has been provided in FWA Loader-4 which points to the first CM Loader table so that the CP Loader tables look like the GPSL ones.

12.7.2 Organization

CPLOADER includes five CP programs and two PP programs which are named:

LOADERQ
LOADERS
LOADERV
LOADERE
MAPOUT
LDQ (PP program)
LOQ (PP program)

Further, in user call mode, the following programs of the standard GPSL are used to load absolute overlays:

OVERLOD (or OVERLOG)
LDR, 2LA, 2LE

12.7.2.1 LOQ

The PP program LOQ is called by 2TS when a loader control card is detected and if the new CP loader flag is set in the control point area. Also, LOQ is called by LOADERQ to reload itself if the checksumming fails.

If the program to be loaded is an overlay of the library, LOQ loads it in CM and enters this overlay.

Otherwise, LOQ looks at the flag "LOADER already-in". If LOADERQ is not in CM or if the checksum of the checksum routine in LOADERQ does not agree, LOQ loads it from CM or makes a request stack to load LOADERQ by one disk access only. When this request is completed, CP is initiated at this control point to execute a short CP program included in LOADERQ (CPLOAD). This program, like a bootstrap, relocates LOADERQ into the highest addresses of the CM user's area, then LOADERQ is entered.

12.7.2.2 LOADERQ

This program has to relocate TEXT tables and builds the CM LOADER tables; it calls CIO or LDQ to read the file. When the physical loading of the file is over, LOADERQ achieves the loading process. The linking of all inter program references is completed. Then, in load card mode, an END request is placed into RA+1 and MTR advances to the next control card. If there is to initiate the execution of the user's program (EXECUTE OR PROGRAM call card), unsatisfied references are filled by loading appropriate library programs. A memory map is produced if it is required by the user, and control is transferred to the specified entry point of the loaded program. This part of the loading process, when the physical load is achieved is the same as the standard GPSL, except the library loading.

12.7.2.3 OVERLAYS OF CPLOADER

When a core map is required, the overlay MAPOUT is loaded into the circular buffer and the MAP is written out.

The "relocatable overlay" LOADERS is loaded in segmentation mode only. This program is the part of standard GPSL which performs all necessary functions for segmentation.

Not any modification has been implemented in this part, except a checksumming. The "relocatable overlay" LOADERV is loaded in overlay generation mode only.

This program is the part of standard GPSL which performs all necessary functions for overlay generation.

No modification has been implemented in this part.

LOADERS and LOADERV cannot be used in the same load. Each of them, when necessary is loaded in the user's area between the CM loader tables and the circular buffer.

LOADERE is an overlay of CP LOADER which is loaded and executed each time a fatal error is detected by any CPLOADER programs (LDQ, LOADERQ, LOADERS, LOADERV, MAPOUT).

This program sends an error message and aborts the job in control card mode or return to the user's program in user call mode.

12.7.2.4 COMMUNICATIONS BETWEEN CP PROGRAMS OF CPLOADER

Some data and subroutines are used by several CP LOADER programs, mainly by LOADERQ, LOADERS, LOADERV, MAPOUT.

For this reason, a communication area is provided as header of LOADERQ program. This area includes a data block, a jump block and a return jump block. In order to ensure the right addressing at the assembly time, Q FIRST has been chosen as an addressing base for the CP programs. Four parameters: LOADSLH, LOADVLH, QBUFL, PARSVE are declared with the same value in the four CP programs, checking are made for that at assembling time and at loading time.

The data block includes all flags and parameters used by all CP LOADER programs.

The jumps block includes all the jumps used to return from LOADERS or LOADERV to LOADERQ.

The return jumps blocks includes the return jump entries to the subroutines used by LOADERS, LOADERV, LOADERQ and MAPOUT.

Any modification in this area must be reproduced in LOADERS, LOADERV and MAPOUT.

12.7.2.5 LDQ

This PP program performs the following functions:

- requests stack processor for high speed reading,
- requests stack processor for library reading,
- reads library programs in CM,
- reads directory from CM or from the disk,
- reads time,
- resets FST,
- processes loader directives.

An FET is used to communicate between LDQ and LOADERQ.

12.7.3.0 INPUT - OUTPUT

There are 3 subroutines for reading in LOADERQ:

- HSPREAD (high speed read)
- NORREAD (normal read)
- LIBREAD (library read).

HSPREAD and LIBREAD cll LDQ, NORREAD calls CIO.

In any case the address of the program to be used is set in QFREAD. High speed read is used in normal loading mode to load a file in relocatable format, if the device is allocatable.

Normal read is used in case of: segmentation, overlay generation, search file and absolute overlay loading. Also normal read functions is used each time the device is non allocatable.

Library read function is used only to read the library.

12.7.3.1 HIGH SPEED READ

LOADERQ sends a request with automatic recall. The CPU is dropped and then LDQ restarts CPU when one sector has been written in the buffer, by deleting the automatic recall address and requesting the CP. In order to avoid spending too much CP time in waiting for the disk, the following organization has been chosen:

When the buffer is empty and the file busy, CP loader waits one sector reading time and then if the buffer is always empty CP LOADER sends a RCL request, because one disk revolution has been lost.

12.7.3.2 NORMAL READ

LOADERQ sends a CIO request with automatic recall so that CPU is dropped and restarted when the physical reading is completed.

12.7.3.3 LIBRARY READ

In this mode, the file status is irrelevant. LDQ PP is waiting during the library read, it sends a stack request each time there is room in the buffer. It indicates the end of the library read by setting a flag in the LDQ reply byte.

12.7.3.4 CIRCULAR BUFFER

These 3 functions use a circular buffer, the length of which can be adjusted (we have chosen QBUFL = 1001_8). The buffer parameters are stored in the QFET, QFIRSTA, QINA, QOUTA, QLIMITA.

12.7.3.5 SPECIAL INPUT

LDQ is requested to read with special function in order to read the tables of the directory and the overlays of CP LOADER.

In these cases a short FET is used and the length of the buffer is determined so that the table or the program can be read with one request stack only.

If the library routine or the table directory is CM resident, LDQ performs the move in CM.

12.7.3.6 OUTPUT

The only output are to write out one absolute overlay or the map. In any case CIO is used with automatic recall. In case of absolute overlay, the overlay itself is considered as a circular buffer. In case of mapping, we choose for buffer all the available space in the user's field length, between CORNEXT and TBLNEXT.

12.7.3.7 WHICH PROGRAMS ARE USED TO READ BINARY FILES

a. CIO in the following cases:

Segmentation mode
Overlays generation
Selective load
Absolute overlay in control card mode
Any kind of loading from magnetic tapes except absolute overlays in user call mode.

b. LDQ is used in the following cases:

Library loading
Normal loading (No segmentation, no absolute overlays,no selective load, no overlay generation)

c. LDR is used in the following cases:

Absolute overlays in user call mode

Consequently, if absolute overlays are recorded on tape, this file is copied onto disk first.

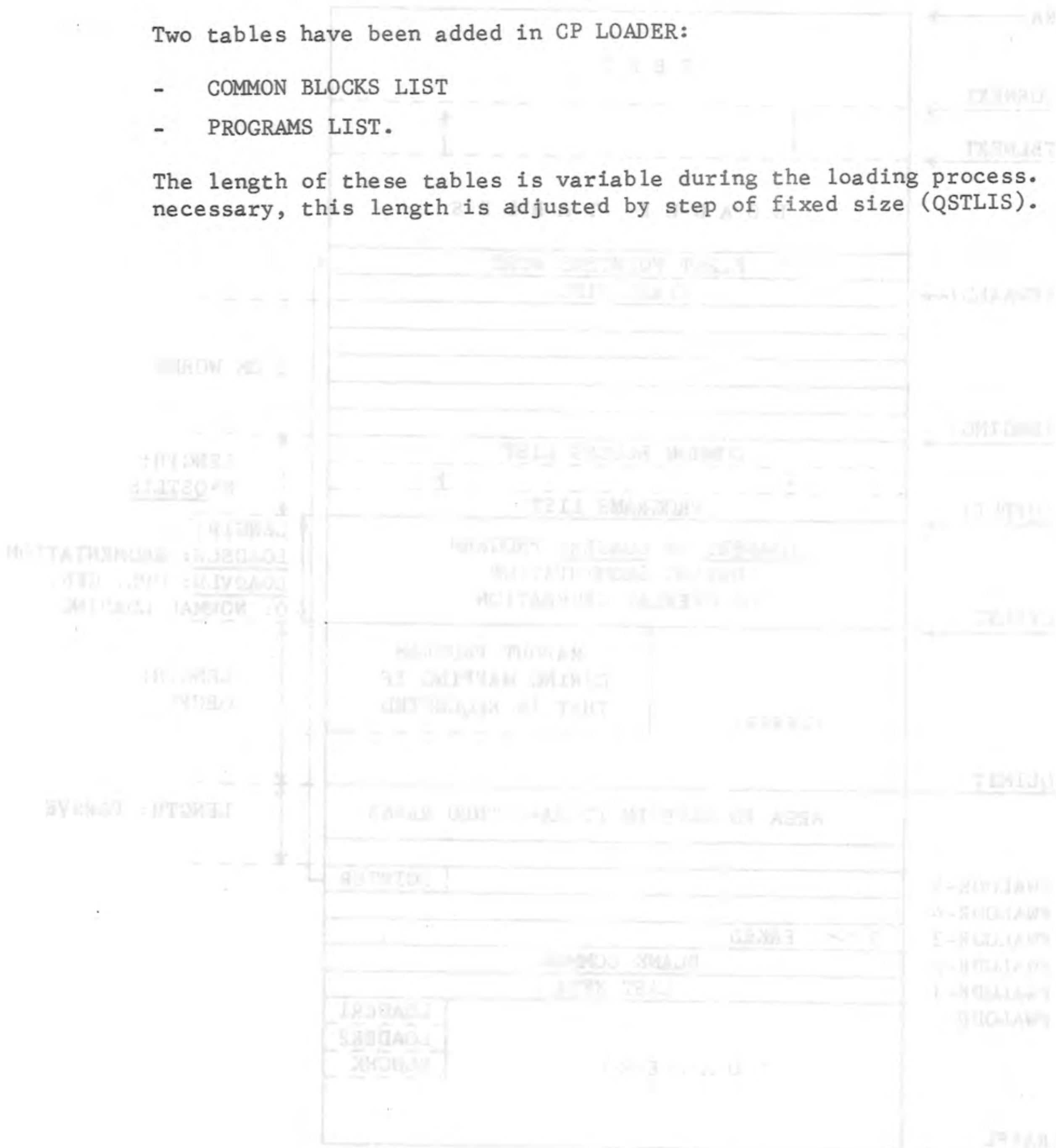
12.7.4.0 FORMATS

The CM loader tables have the same structure as the GPSL ones.

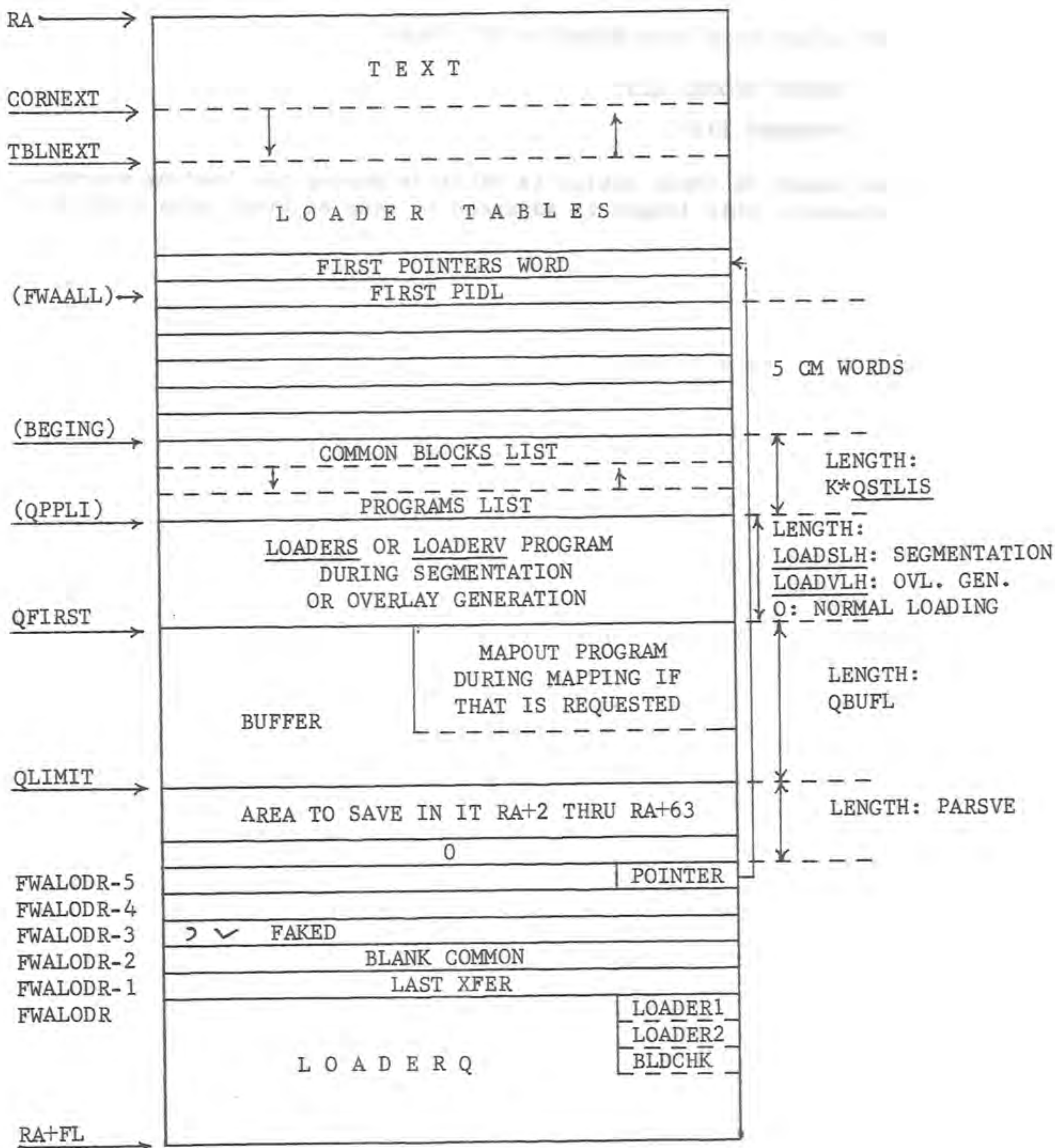
Two tables have been added in CP LOADER:

- COMMON BLOCKS LIST
- PROGRAMS LIST.

The length of these tables is variable during the loading process. If necessary, this length is adjusted by step of fixed size (QSTLIS).



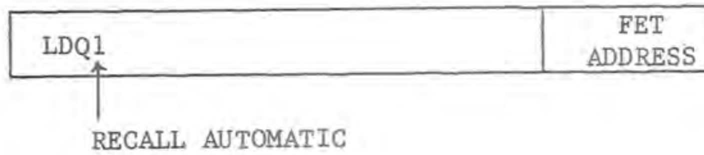
12.7.4.1 CM CORE IMAGE DURING LOADING BY CP LOADER



12.7.4.2 FET FORMAT FOR COMMUNICATION BETWEEN LDQ AND CP LOADER PROGRAMS

QFETA	FILE NAME	FUNCTION AND STATUS
QFIRSTA	1	QFIRST
QINA		QIN
QOUTA		QOUT
QLIMIT		QLIMIT
QFONCA	LDQ REPLY STATUS FLAG	ADDRESS OF PARAMETER
	F S T	

12.7.4.3 CALL FORMAT FOR LDQ



12.7.4.2 FET FORMAT FOR COMMUNICATION BETWEEN LDQ AND CP LOADER PROGRAMS

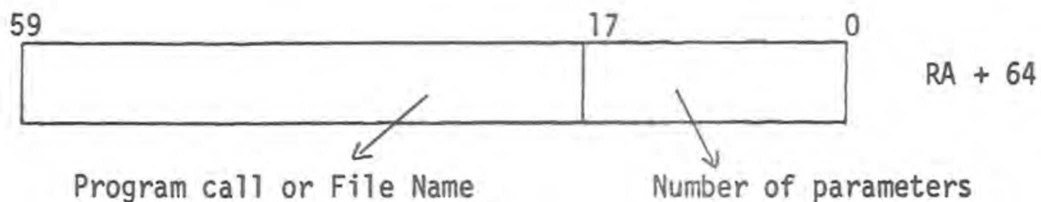
QFETA	FILE NAME	FUNCTION AND STATUS
QFIRSTA	1	QFIRST
QINA		QIN
QOUTA		QOUT
QLIMIT		QLIMIT
QFONCA	LDQ REPLY STATUS FLAG	ADDRESS OF PARAMETER
	FST	

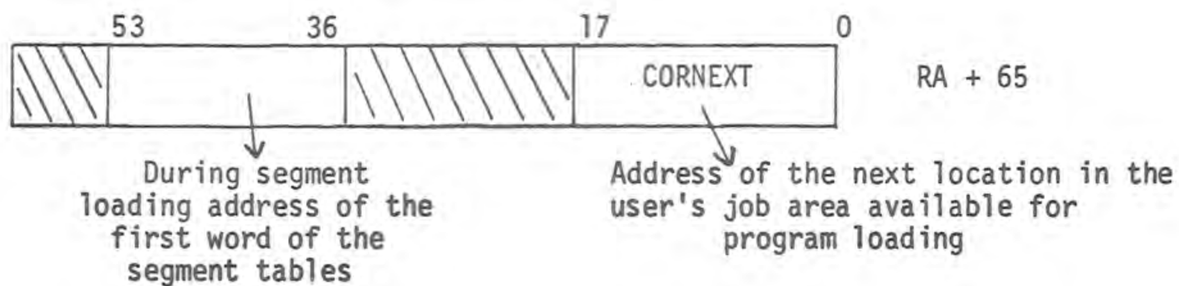
12.7.4.3 CALL FORMAT FOR LDQ

LDQ1	FET ADDRESS

↑ RECALL AUTOMATIC

12.7.4.4 USER COMMUNICATION AREA (RA+64 thru RA+67)





12.7.4.4 USER COMMUNICATION AREA (RA+64 thru RA+67)

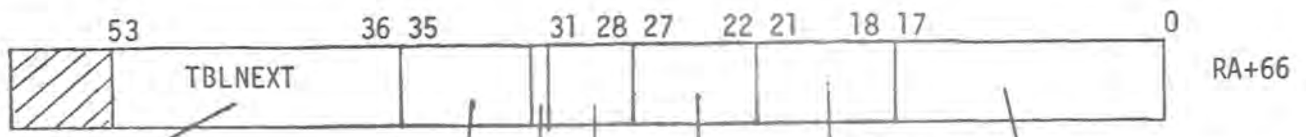
RA + 64

Bits 0–17	Bits 17–59
Number of parameters	Program call or File Name

RA + 65

Bits 0–17	Bits 17–36	Bits 36–53	Bits 53–59
CORNEXT	(hatched)		(hatched)

- CORNEXT: Address of the next location in the user's job area available for program loading
- Bits 36–53: During segment loading address of the first word of the segment tables



Address of the next location available for the loading of tables accompanying LOADER in CM. As the tables are loaded adjacent to and below LOADER, successive locations for loading are obtained by negative addressing from this pointer.

Address of the first word of the object program in the user's job area. This pointer is not utilized by the GPSL routines and is provided as a service to the user.

Bit positions 18, 19, 20, 21 are set when LDR detects the loader directiv cards SECTION, SEGMENT or OVERLAY respectively

OVERLAY flag bits giving the level of incoming overlay. The level of current overlay force writing of overlay.

DEBUG control bits

28 = 1 if labeled dump requested

29 = 1 if change dump requested

30 = 1 if TRACE is to be used with overlay or segment job

31 = 1 if SNAP is to be used with overlay or segment job

32 : = 1 if REDUCE Field Length is to take place

33-35 : Map option bits

001 ON (full map)
010 OFF (no map)
100 PART (partial map)

RA+66

0–17	18–21	22–27	28–31	32	33–35	36–53	
						TBLNEXT	

Bits 0–17: Address of the first word of the object program in the user's job area. This pointer is not utilized by the GPSL routines and is provided as a service to the user.

Bits 18–21: Bit positions 18, 19, 20, 21 are set when LDR detects the loader directly cards SECTION, SEGMENT or OVERLAY respectively.

Bits 22–27: OVERLAY flag bits giving the level of incoming overlay. The level of current overlay force writing of overlay.

Bits 28–31: DEBUG control bits

28 = 1 if labeled dump requested

29 = 1 if change dump requested

30 = 1 if TRACE is to be used with overlay or segment job

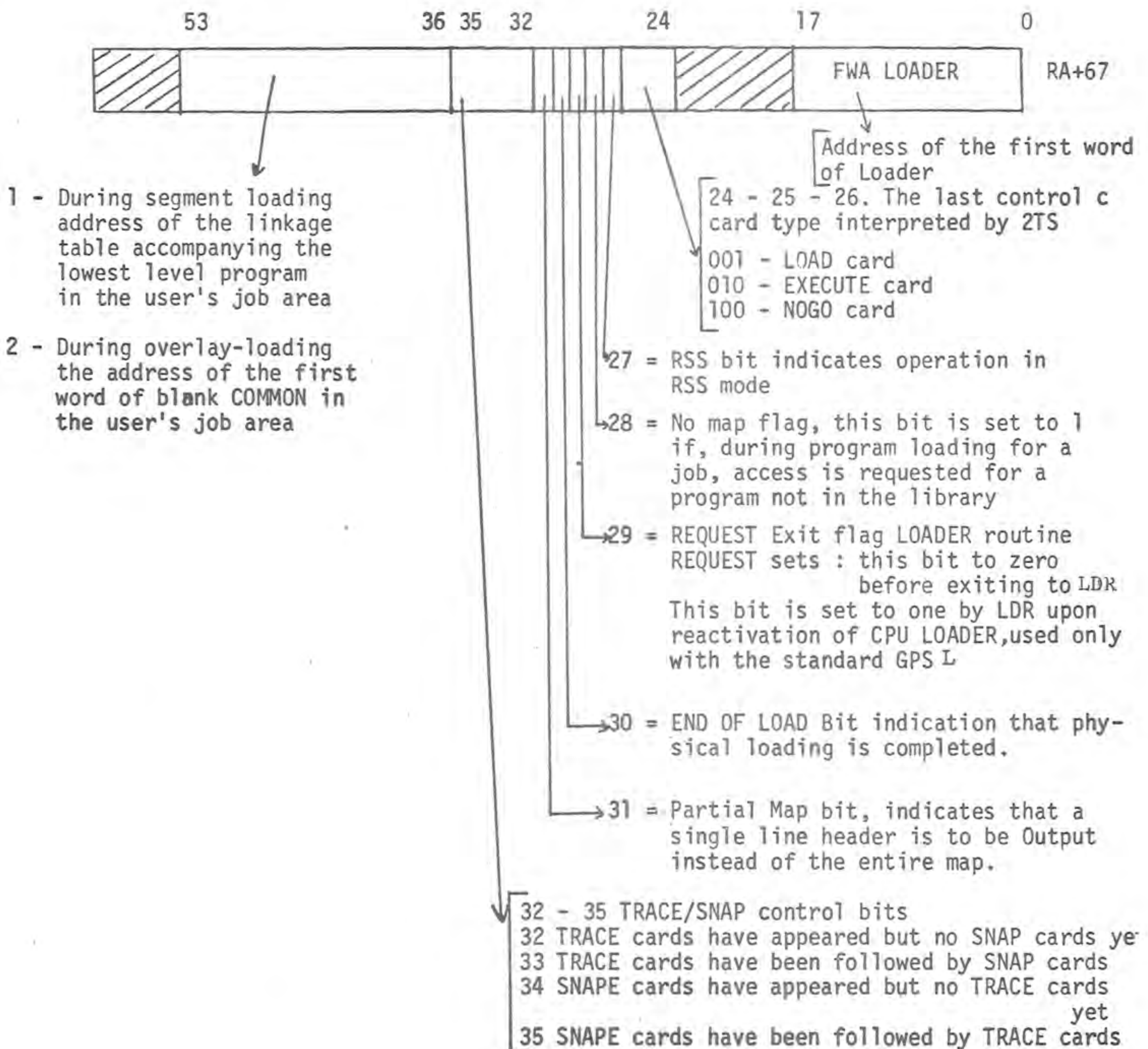
31 = 1 if SNAP is to be used with overlay or segment job

32 : = 1 if REDUCE Field Length is to take place

33-35 : Map option bits

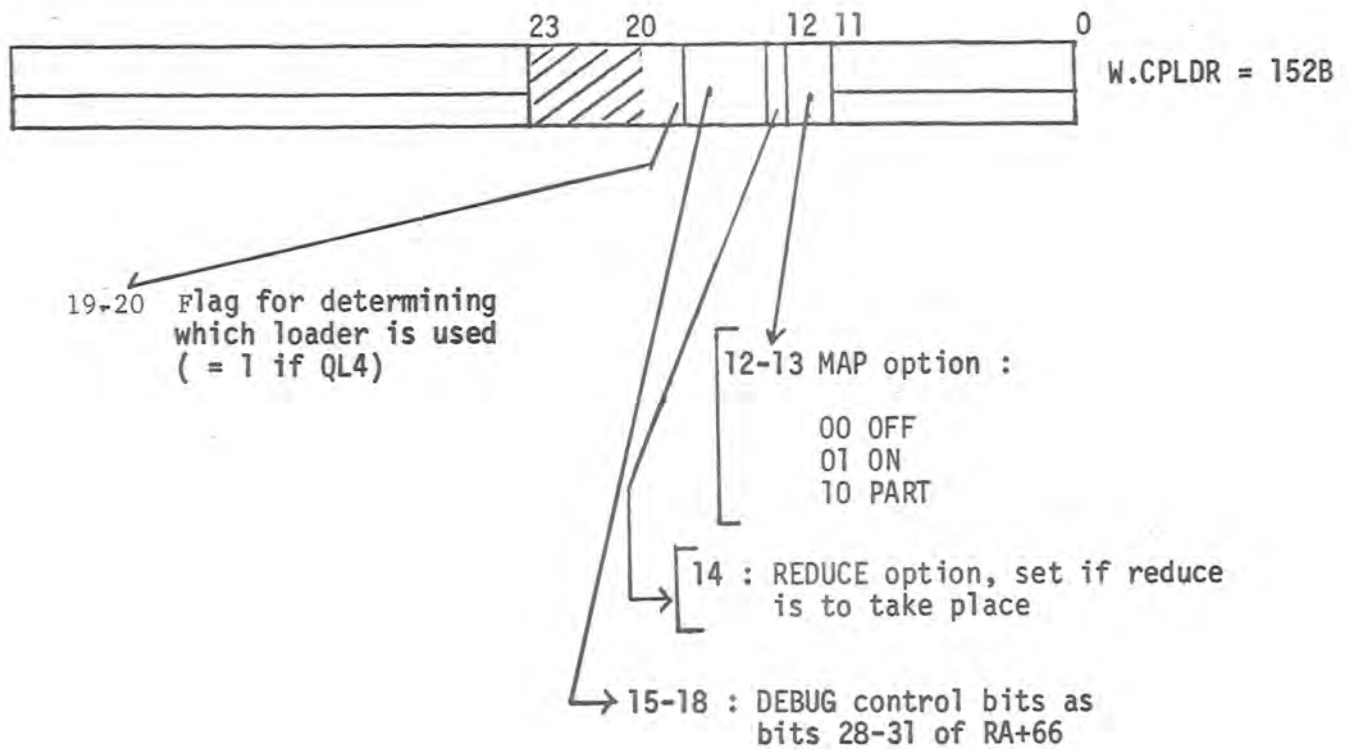
001	ON (full map)
010	OFF (no map)
100	PART (partial map)

TBLNEXT: Address of the next location available for the loading of tables accompanying LOADER in CM. As the tables are loaded adjacent to and below LOADER, successive locations for loading are obtained by negative addressing from this pointer.



53
36 35 32
24
17
0
FWA LOADER
RA+67
Address of the first word of Loader
24 - 25 - 26. The last control c card type interpreted by 2TS
001 - LOAD card
010 - EXECUTE card
100 - NOGO card
1 - During segment loading address of the linkage table accompanying the lowest level program in the user's job area
2 - During overlay-loading the address of the first word of blank COMMON in the user's job area
27 = RSS bit indicates operation in RSS mode
28 = No map flag, this bit is set to 1 if, during program loading for a job, access is requested for a program not in the library
29 = REQUEST Exit flag LOADER routine REQUEST sets : this bit to zero before exiting to LDR
This bit is set to one by LDR upon reactivation of CPU LOADER,used only with the standard GPS L
30 = END OF LOAD Bit indication that physical loading is completed.
31 = Partial Map bit, indicates that a single line header is to be Output instead of the entire map.
32 - 35 TRACE/SNAP control bits
32 TRACE cards have appeared but no SNAP cards ye
33 TRACE cards have been followed by SNAP cards
34 SNAPE cards have appeared but no TRACE cards yet
35 SNAPE cards have been followed by TRACE cards

12.7.4.5 - LOADER FLAG IN CONTROL POINT AREA



12.7.4.5 - LOADER FLAG IN CONTROL POINT AREA

W.CPLDR = 152B

0 | 11 12 | 20 | 23

19-20 Flag for determining which loader is used (= 1 if OL4)

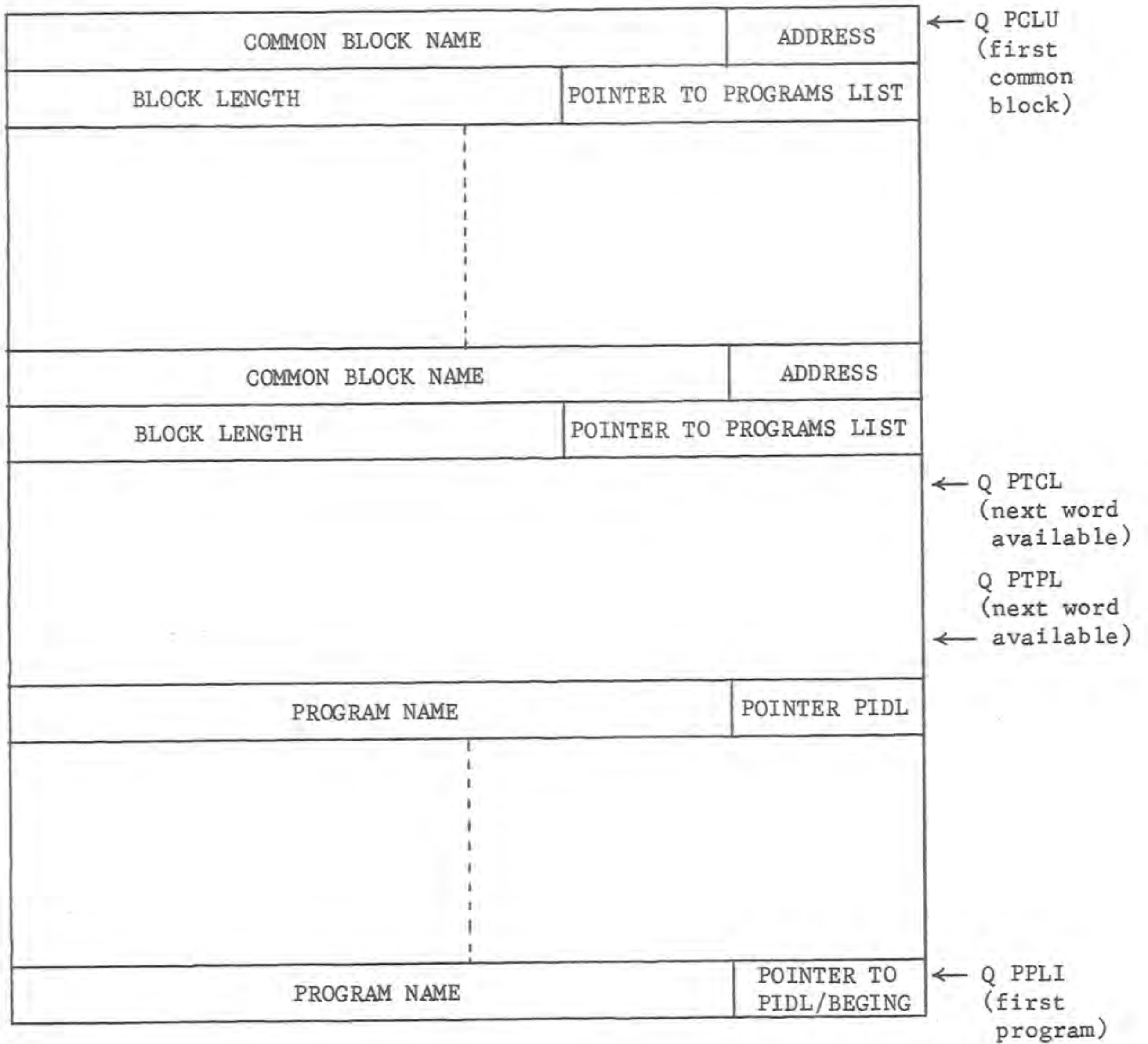
12-13 MAP option :

00 OFF
01 ON
10 PART

14 : REDUCE option, set if reduce is to take place

15-18 : DEBUG control bits as bits 28-31 of RA+66

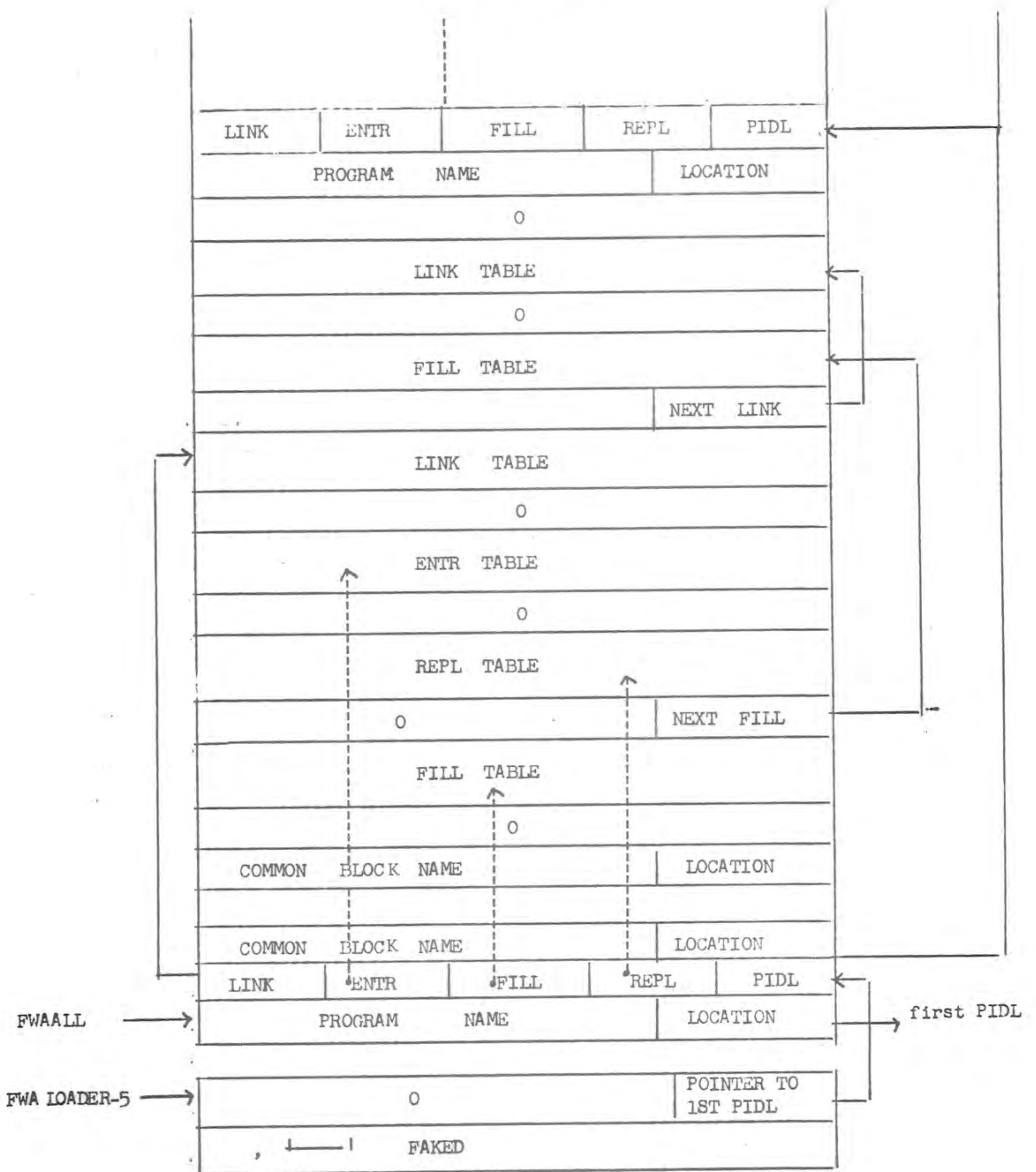
PROGRAMS/COMMON BLOCK LIST



PROGRAMS/COMMON BLOCK LIST

0 PCLU (first common block) →	COMMON BLOCK NAME	ADDRESS
	BLOCK LENGTH	POINTER TO PROGRAMS LIST
	COMMON BLOCK NAME	ADDRESS
	BLOCK LENGTH	POINTER TO PROGRAMS LIST
0 PTCL (next word available) →		
0 PTPL (next word available) →		
	PROGRAM NAME	POINTER PIDL
0 PPLI (first program) →	PROGRAM NAME	POINTER TO PIDL/BEGINC

LOADER TABLES



LOADER TABLES

LINK	ENTR	FILL	REPL	PIDL
PROGRAM NAME				LOCATION
0				
LINK TABLE				
0				
FILL TABLE				
				NEXT LINK
LINK TABLE				
0				
ENTR TABLE				
0				
REPL TABLE				
0				NEXT FILL
FILL TABLE				
0				
COMMON BLOCK NAME				LOCATION
COMMON BLOCK NAME				LOCATION
LINK	ENTR	FILL	REPL	PIDL
PROGRAM NAME				LOCATION

PWAALL → PROGRAM NAME

first PIDL

0	POINTER TO 1ST PIDL
PAKED 1	

PWA LOADER-5 → 0

12.7.5.0 LOQ

12.7.5.0.1 Task description

LOQ is a PP program called either by 2TS or LOADER to load or reload the CP LOADER for a given job.

12.7.5.0.2 Organization and flow analysis

LOQ determines first whether it is called by 2TS or LOADER by checking its input register (1AJ if called by 2TS). Then LOQ determines if it is a control card or program call type.

If program call type, LOQ determines if the call is for a CP program by searching for the program name first in the FNT then in the directory. If found, the CP program is loaded and executed.

If not found, a PP program is assumed.

In case of control card type, or CP program call, the CP LOADER is loaded if the loader already-in switch is not on, or if it is on and the loader checksum routine is not completely in.

If LOADERQ is on an allocatable device, LOQ enters a stack request. In order to read the whole relocatable loader with one stack request, LOQ sets parameters of a CM circular buffer into user's field length (the actual size of this buffer is 5000B but this value can be adjusted). After the stack request is entered, LOQ sets in the control point area the automatic recall pointer to FWA of buffer FET, resets some flags from control point are (W.CPLDR) to user's field length (RA+65,66,67) then exits by activating CP and dropping PP. If LOADERQ is CM resident, LOQ itself reads LOADERQ from the resident area and writes it into the user's field length.

12.7.5.0.3 Subroutines

Open Subroutines

LOD	Initializations	5.1
PPC1	Process program call	5.2
DTL1	Determine to load Loader	5.3
SFL1	Search for Loader	5.4
PPP1	Process PP program call	5.5
CLD1	Read Loader from disk	5.6
RPL1	Read Loader from CM	5.7

Closed Subroutines

SRC	Search directory	5.9
EPT	Search entry point table	5.10
CMP	Compare names	5.11
CKS	Loader checksum routine	5.12
CLEX	Clear exchange jump area	5.13
CNV	Convert octal display code to binary	5.14
GETFWA	Get FWA Load	5.15
TESTFL	Test if sufficient place into FL for Loader	5.16
SETFET	Create a FET in the job field length	5.17
LIP	Store parameters into RA+65,66,67 and process debug cards	5.18

12.7.5.1 LOD

12.7.5.1.1 Task description

This routine initiates LOQ processing.

12.7.5.1.2 Environment description

The PP input register must contain either 1AJ if called by 2TS, or LOQ if called by LOADERQ or RUN.

12.7.5.1.3 Subroutine structure

After reading RA+64 and RA+67 a check is made for LOQ in input register. If LOQ, we do not clear exchange jump area, we drop CP. If called from RUN, LOQ exits to SLF1 after setting a flag for LOADER1 entry.

LOQ determines control card type. If program call,LOQ exits to PPC1. If not, LOQ exits to DTL1.

12.7.5.1.4 Other routines referenced

Exit to SFL1, PPC1, DTL1.

12.7.5.2 PPC1

12.7.5.2.1 Task description

Process program call.

12.7.5.2.2. Environment Description

Routine called by LOD in case of program call type. The program name searched is in RA64. If program name is found in FNT or directory, PPC1 exits to DTL1. If no match is found, PPC1 assumes program name is a PP program and exits to PPP1.

12.7.5.2.3 Other Routines Referenced

CMP Compare names
SRC Search directory
EPT Search entry point table

12.7.5.3 DTL1

12.7.5.3.1 Task Description

Determine to load LOADER.

12.7.5.3.2 Environment Description

The first instruction (RJM, CLEX) is not executed if LOQ is in the input register.

12.7.5.3.3 Subroutine Structure

After (or not) clearing exchange jump area we sense the LOADER ALREADY-IN switch. If LOADERQ is not on, we exit to SFL1. If on we enter CKS to checksum the loader-checksum routine. If not OK, we exit to SFL1. If OK we exit to CP after resetting same parameters (routine LIP).

12.7.5.3.4 Other Routines Referenced

SFL1 Search for LOADERQ
CLEX Clear exchange jump area
CKS Checksum routine
LIP Set parameters

12.7.5.4 SFL1

12.7.5.4.1 Task Description

This routine searches LOADERQ in directory.

12.7.5.4.2 Subroutine Structure

After setting LOADERQ for test, we call SRC to search directory. If LOADERQ is not found, we abort control point, if not we exit to CLD1 if disk resident, or to RPL1 if CM resident.

12.7.5.4.3 Other Routines Referenced

SRC Search directory
CLD1 Load LOADERQ from disk
RPL1 Load LOADERQ from CM

12.7.5.5 PPP1

12.7.5.5.1 Task Description

Process PP program call.

12.7.5.5.2 Environment Description

On entry, we find in NAME a character chain which must be a PP program name. If there are no more than 3 characters for name and two parameters, we form a PP input register. We enter this program in this PP by a LJM R.IDLE.

12.7.5.5.3 Other Routines Referenced

CNV converts a parameter to octal.

12.7.5.6 CLD1

12.7.5.6.1 Task Description

Read LOADERQ from disk in unrelocated form into a CM buffer within user's field length.

12.7.5.6.2 Subroutine Structure

If there is sufficient place in field length, we set a FET at location RA+FL - FWALOAD-10B. This FET defines a buffer with FIRST = RA+FL - FWALOAD and LIMIT = RA+FL. Then we enter a stack request in order to read LOADERQ. We set the auto-recall pointer in control point area to enable to start CP after completion of read operation and to drop PP without waiting for end of LOADERQ read.

12.7.5.7 RPL1

12.7.5.7.1 Task Description

Read LOADERQ from CM in unrelocated form into a CM buffer within user's field length.

12.7.5.7.2 Subroutine Structure

It is the same as CLD1. We do not enter a stack request, but we make a CM - PP - CM transfer.

12.7.5.9 SRC

12.7.5.9.1 Task Description

Search a program name in directory.

12.7.5.9.2 Environment Description

On entry, the name of program to be searched is in direct cells NAME. Exit with A = 0, if no match found:

- A negative; if match and disk resident
- A positive; if match and CM resident, A = CM address of program

12.7.5.9.3 Other Routines Referenced

CMP compare names.

12.7.5.10 EPT

12.7.5.10.1 Task Description

Search entry point table for a name.

12.7.5.10.2 Environment Description

On entry we have a program name in NAME. Exit with (A) = 0 if no match; (A) $\neq$ 0 if match.

12.7.5.10.3 Other Routines Referenced

CMP compare names.

12.7.5.11 CMP

12.7.5.11.1 Task Description

Compare names of seven characters.

12.7.5.11.2 Environment Description

On entry names are in NAME and D.TO. Exit with (A) = 0 indicates a match.

12.7.5.12 CKS

12.7.5.12.1 Task Description

This subroutine computes the 12 bit checksum of LOADERQ checksum routine.

12.7.5.12.2 Environment Description

The length of LOADERQ routine is found in 4th byte of RA+FL-2. Exit with checksum in A.

12.7.5.13 CLEX

12.7.5.13.1 Task Description

Clear exchange jump area store FL in A0 clear RA+1.

12.7.5.14 CNV
12.7.5.15 GETFWA
12.7.5.16 TESTFL
12.7.5.17 SETFET

} self explanatory

12.7.5.18 LIP

12.7.5.18.1 Task Description

Store parameters into RA+65,66,67 and process debug and Reduce cards.

12.7.5.18.2 Subroutine Structure

Flags are read from W.CPLDR of control point area and are copied into user's field length.

TRACE and SNAP cards are processed. Then we reset P in control point area at its correct value and we exit to CP after dropping PP.

12.7.6.0 LDQ

12.7.6.0.1 Task Description

LDQ is a PP program which accomplishes service functions for CP LOADER. The CP LOADER requests are the following:

00B - Read Library Programs into CM buffer

02B - Read non-stop (file load)

04B - Write FST (reset FST at initial value)

06B - Get time

10B - Process Loader directive

12B - Read Library tables (EPT and ERT if any)

14B - Read LOADERQ overlays.

12.7.6.0.2 Organization and flow analysis

LDQ is called only by LOADERQ always with Automatic-recall. The LDQ input register contains a FET address right justified. All parameters are set in this FET by LOADERQ.

	59			17	0	
FET	FILENAME				CODE (STATUS)	
	0 ——— 1				FIRST	+1
	0 ——— 0				IN	+2
	0 ——— 0				OUT	+3
	FNTADR	0 ——— 0			LIMIT	+4
QFLAG	ERROR FLAG	STATFLAG	LASTENT	00	ADDRESS	+5
QFST	FIRST / SECOND] FST WORD					+6

The requested function is stored by LOADERQ in CODE/STATUS. After completion of request LDQ, CIO or OPE sets the status odd and LOADERQ is restarted by MTR.

If an error is detected by LDQ, the error flag is set and LOADERQ will send a dayfile message and abort control point.

12.7.6.0.3 Subroutine

Open Subroutines (functions)

HSREAD	Non stop read	6.1
WRFST	Write FST	6.2
TIME	Get time	6.3
RDLDOVL	Read LOADERQ overlays from library	6.4
EPTERT	Read Entry point and external reference tables	6.5
READLIB	Read Library programs into CM buffer	6.6
LODDIR	Process Loader's directives	6.7

Closed Subroutines

FNT	Search a file name in FNT	6.8
RSLCLD	Search Library directory for a name	6.9
CMPPCM	Transfer CM to CM by PP memory	6.10
STKREQ	Set pointers and enter request in stack	6.11
SYSTEM	Search file name SYSTEM or SSSSSSU and wait not busy	6.12
GETPN	Get next program number	6.13
READDIR	Read into CM buffer a Library program	6.14

The following subroutines are not described here (self - explanatory):

INTOLM	Compute buffer length from IN to LIMIT
PAUS	Pause for relocation - Test error flag
READFET	Read FET of LOADERQ into PP memory
CMP	Compare names of seven characters
RSFET	Reset CMFET from PP memory
GETEQOR	Get equipment ordinal (allocatable device)
GETRBT	Read a RBT entry
GETRBA	Form RBT address
BUFCAP	Compute buffer length
RDLIBP	Read Library pointer
SETDIR	Set pointers in stack request
SETCM	Set pointers in PP direct cells
OUTMIN	Compute OUT minus IN
WAIT	Wait a few milliseconds
RSFLAG	Reset Flag word of FET

12.7.6.1 HSREAD

6.1.1 Task Description

This subroutine has two purposes:

a. the first time, the FET filename is searched in FNT and directory. If found in FNT, we go to b. If found in directory, we set the library load flag for LOADERQ and exit. If filename is not found, we abort the job.

b. we have found the filename in FNT. If the file is on an allocatable device we check load type for rewind and then we enter a special stack request (non EOR-stop Read). If the file is on a non allocatable device, we make an OPEN-REWIND (call of OPE in this PP).

6.1.2 Subroutine Structure

a. If FNT address is not zero, it is not the first call and we go to HSREAD1. We check that we have a good FNT address and if correct we enter the special "non stop read" request in stack. In order to start CP before completion of request (to enable a wrap around

into CM buffer), we are watching FET IN/OUT pointers. After the read of the first PRU into CM buffer, we clear the auto - recall pointer in control point area and we restart CP before exiting from PP.

b. FNT address is zero. It is the first time we are called.

Name is not in FNT, nor in directory = abort control point.

Name is found in directory = set flag for LOADERQ and exit.

Name is found in FNT:

- if non allocatable device we load OPE in this PP for OPEN-REWIND;
- if allocatable device, we check for rewind (no rewind if name = INPUT or selective file Load) and we go to a.

6.1.3 Other Routines Referenced

FNT	Search FNT for a name
RSLCLD	Search directory
REWIND	Execute a rewind or call OPE
CMP	Compare names
RSFET	Reset FET (new buffer status)
GETEQOR	Get equipment ordinal
STKREQ	Complete and enter stack request
WAIT	Wait a few milliseconds
READFET	Read CM FET
OUTMIN	Compute OUT minus IN
RSFLAG	Reset error flag word

12.7.6.2 WRFST

6.2.1 Task Description

Reset FST of a file on an allocatable device to its old value.

6.2.2 Subroutine Structure

Before we re-write the old FST, we make some checks to be sure we cannot bring some trouble to the operating system.

12.7.6.3 TIME

6.3.1 Task Description

After reading T.CLK we write time at the address set in its FET (word FET+5, last 18 bits) by LOADERQ.

12.7.6.4 RDLDOVL

6.4.1 Task Description

Read from Library LOADERQ overlays (LOADERE, LOADERS, LOADERV, MAPOUT).

6.4.2 Subroutine Structure

First, we search the directory for the name of overlay. If not found, it is a system error. If found, we go to ERTREAD (see EPTERT section).

6.4.3 Other Structure Referenced

RSLCLD search a name within directory.

12.7.6.5 EPTERT

6.5.1 Task Description

Read entry point and external reference table.

6.5.2 Subroutine Structure

First, we read Entry Point Table into a CM buffer. If no space available, we abort job. Then, we look at Program name table. If we find an External reference table, we read it into CM just at the end of EPT. ERT may be in CM or on allocatable device.

6.5.3 Other Routines Referenced

RDLIBP	Read Library pointer
CMPPCM	Transfer CM to CM by PP memory
BUFCAP	Compute buffer length
RSFET	Reset FET
READDIR	Read directory program

12.7.6.6 READLIB

6.6.1 Task Description

Read Library programs into CM buffer. Address of a CM list of program numbers is in QFLAG.

6.6.2 Subroutine Structure

We read into PP the first of program number. We call GETPN to obtain the next program number. If zero the list is ended and we set the LOADERQ flag end of Library load.

If non-zero, it is a program number. We get PNT entry after some verifications and if on an allocatable device, we enter a stack request. If core resident we make the transfer.

6.6.3 Other Routines Referenced

RDLIBP	Read library pointer
GETPN	Obtain next program number
SYSTEM	Search file SYSTEM in FNT
STKREQ	Complete and enter stack request
WAIT	Wait a few milliseconds

12.7.6.7 LODDIR

6.7.1 Task Description

Process Loader's directives.

6.7.2 Subroutine Structure

We read the entire Loader's directive which has been detected by LOADERQ, into PP memory. This subroutine is exactly copied onto 2LA overlay of GPSL.

12.7.6.8 FNT

6.8.1 Task Description

Search FNT for a match on a seven characters name.

6.8.2 Environment Description

On entry the name is in direct cells D.FNT,.,D.FNT+4. If a match is found, exit with A register non-zero. If no match, exit with A = 0.

6.8.3 Other Routines Referenced

CMP Compare names.

12.7.6.9 RSLCLD

6.9.1 Task Description

Search entry Point table for a name.

6.9.2 Subroutine Structure

The entry point table is searched for a name, the program name table is not searched for a CP relocatable program or overlay because it has been done by LOQ. If match is found exit with A non-zero.

6.9.3 Other Routines Referenced

RDLIBP Read Library pointer

CMP Compare names

12.7.6.10 CMPPCM

6.10.1 Task Description

Transfer CM to CM by PP memory.

6.10.2 Environment Description

On entry, we must find in direct cells FWR and FWR + 1 the FWA of the input area from CM, in LWR, LWR + 1, LWA + 1 of the same area. Buffer parameters are in FET.

Exit with A = 0, if end of transfer. Exit with A negative if buffer full (FWR, FWR+1 are reset to correct values) IN is updated in FET.

6.10.3 Other Routines Referenced

BUFCAP	Compute buffer length
INTOLM	Compute buffer length from IN to LIMIT.

12.7.6.11 STKREQ

6.11.1 Task Description

Complete (set pointers and flags) and enter request in stack.

6.11.2 Environment Description

On entry, we find in lower 12 bits of A the flags for the request (FNT or not). In upper 6 bits of A, we find the request order (normal read, non-stop read or PP read).

12.7.6.12 SYSTEM

6.12.1 Task Description

Search file name SYSTEM or SSSSSSU in FNT and wait for not busy.

6.12.2 Subroutine Structure

First, we check the PNT entry. If an EDITLIB has been made on this program, we search the common file SSSSSSU. If not, we search the common file SYSTEM. If wo do not find this file, we display the message LDQ SYSTEM WAIT and we try again. When this file is not busy, we read FST entry.

12.7.6.13 GETPN

6.13.1 Task Description

This routine returns the next program number from the LOADERQ's list.

6.13.2 Subroutine Structure

We check a flag to know if we have to read a new word. Then, we use the last index number and we extract from a CM word the first, second, third or last 15 bits. On exit this 15 bits number is in A, right-justified.

12.7.6.14 READDIR

6.14.1 Task Description

Read into CM buffer a directory program.

6.14.2 Subroutine Structure

We have to read a directory program into a CM buffer: if allocatable device resident we enter a stack request, if CM resident we read into buffer by CMPPCM routine. If the buffer is too small, we do not set FET status to EOR.

6.12.2 Subroutine Structure

First, we check the PMT entry. If an EDITLIB has been made on this program, we search the common file SSSSSSU. If not, we search the common file SYSTEM. If we do not find this file, we display the message LDQ SYSTEM WAIT and we try again. When this file is not busy, we read FST entry.

12.7.6.13 GETPN

6.13.1 Task Description

This routine returns the next program number from the LOADERQ's list.

6.13.2 Subroutine Structure

We check a flag to know if we have to read a new word. Then, we use the last index number and we extract from a CM word the first, second, third or last 15 bits. On exit this 15 bits number is in A, right-justified.

12.7.6.14 READDIR

6.14.1 Task Description

Read into CM buffer a directory program.

6.14.2 Subroutine Structure

We have to read a directory program into a CM buffer: if allocatable device resident we enter a stack request, if CM resident we read into buffer by CMPTCM routine. If the buffer is too small, we do not set FET status to EOR.

12.7.7.0 LOADERQ

12.7.7.0.1 Task Description

LOADERQ is a CP program which is loaded in the user's CM area by the PP program LOQ.

There are two main tasks which are accomplished by LOADERQ.

a. First task is to read into the user's CM area the user's programs and the library programs. For this reading, LOADERQ requests the PP programs CIO or LDQ. The tables programs are read into a circular buffer, LOADERQ processes the TEXT table by relocating the addresses; the XFER, LINK, ENTRY, REPLICATION and FILL tables are just linked and put in CM with the same structure as the GPSL one; the "Instant Replication" tables are processed immediately. For each PIDL table, LOADERQ creates a new entry in the tables of of Loader and assigns the CM length to each labelled common block. An absolute overlay table (code 50) is processed as an absolute overlay 0,0, it is written in CM and the control is transferred to this overly. In case of Loader Directive LOADERQ assembles it in a buffer and requests LDQ for processcing;

b. The second task is executed after the physical reading is finished. In this part, FILL tables, LINK TABLES and REPL tables are processed. Except for library loading, all sequences and subroutines used for this second task are the GPSL ones.

c. For the other tasks, LOADERQ needs some "relocatable overlays" of CP Loader:

MAPOUT	to make mapping if it is required;
LOADERS	to process the requests to loader with segmentation
LOADERV	in case of overlay generation to write out absolute overlays
LOADERE	to process fatal errors.

d. LOADERQ initiates the job's process if it is required (control card type).

12.7.7.0.2 Organization and flow analysis

LOADERQ can be entered at the following points:

LOADER	in case of user call, by using a return jump;
LOADER1	in case of user call if the checksum does not agree LOADERQ requests LOQ to reload LOADERQ, and LOQ returns to LOADER1 after loading;
LOADER2	in control card mode, if LOADERQ has just been loaded;

LOADER3 in control card mode, if LOQ has detected that LOADERQ was already in CM;

LOADER4 in control card mode and overlay generation (overlays on different files) if LOQ has detected that LOADERQ was already in CM.

The first binary table of LOADERQ program is an absolute format table (code 50); this table is a short CP program, called CPLOAD which has been assembled in absolute mode. This routine relocates LOADERQ and puts it relocated into the highest addresses of the user's CM area.

In order to accomplish this relocation, CPLOAD relocates itself first, then it links back all tables which constitute LOADERQ and then it relocates and puts them at the right place into CM. Before exit to LOADER1 or LOADER2, CPLOAD builds a 12-bits checksum of the LOADER checksum routine.

After checksumming, jump to CARDCAL is made in case of control card mode, and jump to CALL in user call mode.

a. Control card mode

If there is no user's programs to be read (NOGO card or EXECUTE card) jump is made to CONTROL and the loading process is ended by the same way as the standard GPSL (except for library loading). Otherwise, jump to FNCAL is made. Then physical reading of the file is initiated.

In any case, in control card mode, high speed read function is used first, and it is kept if there is not loader directive.

If a loader directive is found, LDQ PP program is requested to reset the FST of the file and then normal read function will be used by calling CIO.

In any case, if the file to be loaded is on a non allocatable device, normal read function is used with CIO.

In normal loading mode (no loader directive) the CN code of each table is looked at and the appropriate process is executed.

The physical loading process goes on until End of File except if an absolute overlay 0,0 is detected, in this case overlay is loaded and entered at the entry address. Otherwise, when the file is loaded, CONTROL routine of standard GPSL is entered.

If a loader directive is detected, LOADERQ assembles it in a buffer and calls LDQ PP program to process it.

In case of segmentation, all loader directives are processed and when the first binary table is detected, the file is backspaced and CONTROL routine is entered; segmentation mode is selected by this routine and the "relocatable overlay" LOADERS is loaded and entered in order to make a call for segment zero.

In case of overlay loader directive, the loading process of the overlay's file is stopped at each loader directive (and at EOF). Each time, CONTROL routine is entered, this routine detects the overlay generation process, it loads the "relocatable overlay" LOADERV if it is not yet in CM and gives it the control in order to achieve the loading process and to write out the last overlay if it is necessary.

After each overlay, LOADERV calls REQUEST subroutine to load the next overlay.

b. User call mode

Routine CALL is entered in case of user call. First, the flags of the user's call are processed. For segmentation mode, the control is transferred to LOADERS, "relocatable overlay" of Loader. This overlay is already loaded but checksumming is made first.

For non-segmented loads, the routine CALLOK is entered. This routine calls subroutine REQUEST by executing a return jump, in order to read the file.

This subroutine looks at the flags-bits of the call. High speed read mode is initiated exept if an SL list address is specified or in case of library loading mode. Then jump to FNCAL is made as for control card mode. When physical loading is completed, REQUEST returns to CALLOK.

c. Library loading

Each time library loading is required, FILLREQ (or SATISFZ) routine is entered. This routine looks for the unsatisfied external references and builds the list of the entry points. Once this list is filled, FILLREQ call LDQ to get the Entry Point table of the library directory and it sets up the list of the numbers of the library programs which are required. Then if there is an External Reference Table and in normal loading mode (no segment no overlay), this routine processes the ERT to set up the list of all library programs which are required and the subroutine REQUEST is entered to load these programs.

12.7.7.0.3 Subroutines

We describe only the specific routines of CP LOADER, see GPSL maintenance documentation for the others.

Open Subroutines

CARDCAL	Initiate processing of load card	7.1
FNCAL	Initiate reading of file	7.2
QPTH1	Look at the CN code and initiate appropriate process	7.3
SLPROC	Process SL list	7.4
UECALPR	Initiate library loading	7.5
QPPIDLT	Process PIDL table	7.6
QPTEXTT	Process Text table	7.7
QPREPLT	Process REPL table	7.8
QPXFERT	Process XFER table	7.9
QPOVLT	Process absolute overlay	7.10
QLOADIR	Process Loader directive	7.11
QPALLT	Process other tables (LINK, ENTRY, FILL)	7.12
CMJUMP	Complete physical loading	7.13
ENOVSG	Enter an overlay of loader for overlay generation or segmentation	7.14
USCLIB	Process user call with library bit (U)	7.15
REDUCE	Reduce field length at the end of loading	7.16
SKPLIB	Skip one library program	7.51

Closed Subroutines

QINITI	Save some registers	7.17
QSINITI	Initializations	7.18
REQUEST	Read file or library, in case of user call	7.19
PUTREQ	Write a PP call in RA+1	7.20
SENDRCL	Make CP recall request	7.21
SCIO	Call CIO	7.22
NORREAD	Normal read routine	7.23
HSPREAD	High speed read routine	7.24
LIBREAD	Read library routine	7.25

REWIND	Rewind a file	7.26
BACKSPC	Backspace a file	7.27
SKIPFB	Skip forward one record	7.28
RESETRD	Reset normal read mode	7.29
RESFST	Reset FST	7.30
QSSCOTB	Store CORNEXT TBLNEXT in CM	7.31
RSETBUF	Reset buffer parameters	7.32
QSGOW	Fetch one word from the buffer	7.33
QSINCPL	Make room in CM for the programs or common blocks list	7.34
CHKPAR	Parity check	7.35
ERRLDQ	LDQ error check	7.36
SEAPIDL	Look for a program in the programs list	7.37
SETOUTL	Set limit of OUT	7.38
LOOVLO	Load an overlay of loader	7.39
RLOVLOD	Relocate an overlay of loader	7.40
MAP	Call MAPOUT overlay for mapping	7.41
TBLSET	Set a relocation bases table	7.42
ROVL	Read an absolute overlay	7.43
SETIN	Initialize flags and parameters	7.44
MOVECM	Move a part of CM in user's field length	7.45
ROOMCM	Make room in user's field length to load an overlay of loader	7.46
WAITFF	Send recall request until a file is not busy	7.47
FILLREQ	Perform library loading	7.48
ERROR	To process fatal and non fatal error	7.50

12.7.7.1 CARDCAL

7.1.1 Task Description

This routine looks at the card call type and initiates the appropriate processing.

7.1.2 Environment Description

This routine is entered in control card mode immediately after checksumming if this is not the first entry in LOADERQ. At the first entry in LOADERQ, subroutine SETIN is executed (CARDCAL1) before CARDCAL is entered. Control cards type bits must be set in RA+67.

7.1.3 Subroutine Structure

First this subroutine makes some initializations by calling QSINITI and RSETBUF, then it saves the parameters (RA+2...) into a CM area referenced PPPPPPP. Some flags are set for LDQ usage. In case of EXECUTE or NOGO card, we exit to CONTROL, otherwise file name is set in the FET and we exit to FNCAL to make physical loading.

7.1.4 Other Routines Referenced

QSINITI	Initialize parameters and flags
RSETBUF	Initialize buffer parameters
SHUFFL1	Save parameters in PPPPPP

12.7.7.2 FNCAL

7.2.1 Task Description

To initiate read file, to save the FST of the file, to set MAPBIT if this is a user program, and to set WRONGF flag to detect an empty file.

7.2.2 Environment Description

This routine is entered in control card and user call mode. The read function code must be set in QFREAD before.

7.2.3 Subroutine Structure

First this routine initiates the read function routine. When the read function is initiated, we return to FNCAL3, except in two cases:

- we have to load a library routine, in this case we execute the subroutine FILLREQ;
- the file is recorded on a non allocatable device, in this case return is to FNCAL4.

This routine is entered in FNCAL4, in overlay generation mode.

When the read function is initiated, in case of user's programs, we exit to QPTH1.

7.2.4 Other Routines Referenced

The read routine to be used (NORREAD or HSPREAD).

12.7.7.3 QPTH1

7.3.1 Task Description

To identify table and to initiate its processing.

7.3.2 Environment Description

CORNEXT must be in B7, TBLNEXT in B6, OUT in B5. If the circular buffer is not empty, OUT must indicate a table header or a loader directive.

7.3.3 Subroutine Structure

First this routine fetches one word from the buffer, and processes it as a header table.

Word count is saved in B4, L in QLSAV, LR in QLRSAV.

Then, we look at the QCNTBL (table of CN codes) and pick the address according to the CN code and the process of the table is initiated. The header table is saved in X5 for later usage.

7.3.4 Other Routines Referenced

QSGOW	To read one word from the buffer and to save it in X2
QSSCOTB	To save CORNEXT and TBLNEXT in CM

12.7.7.4 SLPROC

7.4.1 Task Description

To compare the user's list against the list of the loaded programs in order to determine which programs must be loaded.

7.4.2 Environment Description

This routine is executed in user call mode only, if there is an SL list address.

7.4.3 Subroutine Structure

This routine looks at the user's list and at the program's list in order to determine which programs must be loaded. For each program already loaded, the SL list is updated with its address.

At the end, SLCOUNT indicates the number of programs to be loaded. REWCNT is set for search file with end-around. We exit to FNCAL after normal read function is set.

12.7.7.5 UECALPR

7.5.1 Task Description

To initiate the library loading process.

7.5.2 Environment Description

CORNEXT and TBLNEXT must be set in B7 and B6. The list of the programs number to be loaded is set up in PNTABL, ended by a zero word.

7.5.3 Subroutine Structure

The library read function (LIBREAD) is set in QFREAD. Then CORNEXT, TBLNEXT are saved in CM and LDQ is called to initiate the library read process. Exit is to QPTH1.

7.5.4 Other Routines Referenced

QSSCOTB	To save CORNEXT, TBLNEXT
PUTREQ	To call LDQ

12.7.7.6 QPPIDLT

7.6.1 Task Description

To process a PIDL table of a routine. To initiate processing for a program, including the initiation of a new table subgroup within the CM LOADER tables. COMMON allocation, if any, are saved in the LOADER tables and assigned storage addresses.

7.6.2 Environment Description

CORNEXT	must be set in B7
TBLNEXT	in B6
WORDCOUNT	in B4

7.6.3 Subroutine Structure

There is an error exit if absolute overlay flag is set. If the table is the first one of a file with section or segment process, the file is backspaced and we exit to CMJUMP after setting a "loader directive skip" flag in order to skip later all loader directives on this file during this load.

In user call mode with search file, PIDLSL1 routine is entered; we look at the user's table to determine if this program is to be loaded. If yes, we exit to QPIDL2 to load it, if not, we skip this routine and exit to QPTH1 to process the next one. Before skipping the routine the status of the file is looked at. If EOF, the file is rewound unless it is an INPUT file or the file has been rewound once, in both cases, we exit to CMJUMP.

In any other case, PIDL table is processed.

First, in library loading or in program call card mode, the program's list is looked at in order to determine if this program is already loaded; if yes, it is skipped and a non fatal message is written out in case of program call card.

CORNEXT, relocation base, and address where to put the relocated text are updated. A new table subgroup is initiated within the CM loader tables, and the programs list is updated. Then common blocks are processed if any. For each common block, the common block list is looked for. CM allocation is made for each new labelled common block. For each labelled common block which are already in the list, the new length is checked and a non fatal message is written if the new length is greater than the old one. Then, SL list is updated with the program address if there is a SL list and the routine TBLSET is executed to set the table of the relocation bases to be used in text relocation. Exit to QPTH1 to process next table.

7.6.4 Other Routines Referenced

QSGOW	To fetch one word from the circular buffer
SEAPIDL	To look at the program's list for programs already loaded
QSINCPL	To make room for program's or common lists if there is not enough
QSSCOTB	To save CORNEXT and TBLNEXT in CM
BACKSPC	To backspace the file
SKIPFB	To skip one record
REWIND	To rewind the file
TBLSET	To set up the relocation bases table

12.7.7.7 QPTEXTT

7.7.1 Task Description

Table processor which relocates program text to program area of the users job area. Addresses are relocated as indicated by table relocations bits.

7.7.2 Environment Description

The relocation bases table must be set up in TABLE1.

L (Load address) is in QLSAV
LR (Relocation of load address) is in QLRSAV
Word count is in B4
CORNEXT and TBLNEXT in B7 and B6

7.7.3 Subroutine Structure

First, the load address is relocated and some checking is made for the value of LR and the length of the table. The, we get the relocation control word, we save CORNEXT and TBLNEXT and the relocation loop (QRELOP) is entered.

In this loop, we compute first the number of words to be relocated. This number is the length of the text table or the number of words available in the circular buffer if the text table is not in the buffer in its entirety. Then these n words are relocated in a very efficient stack loop which performs the relocation at about 3 micro seconds/word, by using relocation bases set in TABLE1.

Then the relocation loop (QRELOP) is entered twice if there are still words of this table to be relocated.

When the whole TEXT table is relocated, we exit to QPTH1 to process next table.

7.7.4 Other Routines Referenced

QSGOW	To get the relocation word
QSSCOTB	To store CORNEXT and TBLNEXT in CM
SETOUTL	To set limit of OUT

Read program (NORREAD, HSPREAD or LIBREAD) to fill up the circular buffer.

12.7.7.8 QPREPLT

7.8.1 Task Description

To put REPL table into the CM loader tables or to initiate immediately its processing, if this is an instant REPL table.

7.8.2 Environment Description

Word count in B4
L in QLSAV.

7.8.3 Subroutine Structure

If L equal to zero, this is not an instant replication table, exit is to QPALLT to put this table in CM.

If L is not equal to zero, we have an instant replication table, in this case, B4, B5, B6, B7 registers are saved and the subroutine INSTANT is executed to process this table. Exit is to QPTH1 after all tables of this kind are processed.

7.8.4 Other Subroutines Referenced

QSGOW	To fetch one word
QSSCOTB	To save CORNEXT and TBLNEXT
INSTANT	To process the instant REPL table
QPALLT	To put REPL tables in CM.

12.7.7.9 QPXFERT

7.9.1 Task Description

To save last XFER table (FWA Loader-1) in FWA Loader-3 and to replace it by the new one in FWA Loader-1.

12.7.7.10 QPOVLT

7.10.1 Task Description

To load an absolute overlay 0,0 into the user's field length.

7.10.2 Environment Description

The overlay header must be in X5.

7.10.3 Subroutine Structure

First normal read mode is set if necessary.

Then absolute overlay is loaded into the user's field length by using ROVL subroutine.

In case of program call card or user call, exit is to the entry address of this overlay.

In case of load card, exit is to QPTH1 and the entire file is loaded.

7.10.4 Other Routines Referenced

RESETRD	To set normal read mode
ROVL	To read one absolute overlay
RSETBUT	To reset buffer parameters
QSSCOTB	To store CORNEXT TBLNEXT in CM.

12.7.7.11 QLOADIR

7.11.1 Task Description

To assemble a loader directive into BUFLDR buffer and call LDQ for processing.

7.11.2 Environment Description

First word of Loader directive must be in X5.

7.11.3 Subroutine Structure

First normal read mode is set if necessary. If the overlay generation flag is set, the file is backspaced and exit is to CMJUMP to complete the loading of this overlay.

Otherwise, the loader directive is assembled into BUFLDR buffer; then if the skip flag is set exit is made to QPTH1 to process next table, if it is not set overlay loader directive flag or segment loader directive flag is set according to this loader directive and LDQ is called to process it.

When this process is completed, we look at the "write out" bit in RA+66 and if we have to write the overlay, exit is to CMJUMP after the file is backspaced.

If "write out" flag is not set, exit is to QPTH1 to process next table.

7.11.4 Other Routines Referenced

RESETRD	To set normal read mode
BACKSPC	To backspace the file
QSGOW	To get one word from the circular buffer
PUTREQ	To call LDQ
WAITFF	To wait the loader directive process is finished
ERRLDQ	To check the reply of LDQ.

12.7.7.12 QPALLT

7.12.1 Task Description

To provide entries in CM LOADER tables for LINK,REPL, FILL, ENTR tables.

7.12.2 Environment Description

This routine is entered by QPFILLT, QPREPLT, QPLINKT, QPENTRT. The type of the table is set in B2.

7.12.3 Subroutine Structure

The tables subgroup of the program being loaded is looked at. Entry for this table is initiated in the pointers word if this is the first table of this kind and the table is written into the CM loader table. Exit is to QPTH1 to process next table.

7.12.4 Other Routines Referenced

QSGOW to fetch one word from the circular buffer.

12.7.7.13 CMJUMP

7.13.1 Task Description

This routine is entered when the physical loading is achieved. CORNEXT, TBLNEXT and the loader directives flags are reset in CM. Exit is to CONTROL in case of control card mode. In user call mode, exit is to REQUEST after resetting some register (A0, B2, B4, B5, B6, B7).

12.7.7.14 ENOVSG

7.14.1 Task Description

To load "relocatable overlay" of loader (LOADERS or LOADERV) if necessary and exit to this overlay at a specified entry point.

7.14.2 Environment Description

Entry point in B1.
Length of the overlay (LOADSLH or LOADVLH) in B2.

7.14.3 Subroutine Structure

First this routine determines if the requested overlay is already loaded. If yes, the overlay is entered after checksumming for the overlay LOADERS. If not, or if the checksum of LOADERS fails, the requested overlay is loaded and then the overlay is entered.

7.14.4 Other Routines Referenced

ROOMCM	To make room for the overlay
LOOVLO	To load the overlay
RLOVLOD	To relocate the overlay

12.7.7.15 USCLIB

7.15.1 Task Description

To load programs library, the list of which is provided by user call.

7.15.2 Subroutine Structure

Error exit is made if there is no SL list. Otherwise, the list of the required programs is set up in a buffer of LOADERQ and subroutine FILLREQ is executed to load these programs. After this load, the user's list is updated with the addresses of these programs.

12.7.7.16 REDUCE

7.16.1 Task Description

To release the field length which is not necessary for the execution of the program.

7.16.2 Environment Description

This routine is entered in normal load mode just before initiate the execution of the user's program, if the REDUCE flag is set.

7.16.3 Subroutine Structure

This subroutine determines the new value of the field length according to the blank common length if there is one.

Then a short program is moved in the highest words used and entered. It sends a MEM call in RA+1, waits for the completion, and clears all Ai registers, except A0 which includes the new field length. Control is transferred to the user's program.

12.7.7.17 QINITI

7.17.1 Task Description

To make some initializations in user call mode.

7.17.2 Subroutine Structure

This subroutine saves the following registers: A0, B2, B4, B5, B6, B7 and it calls QSINITI subroutine to initialize some parameters.

12.7.7.18 QSINITI

7.18.1 Task Description

To make some initializations before physical loading.

7.18.2 Subroutine Structure

First Loader directives flags are stored in STATFLAG for LDQ usages, CORNEXT is set in B7 and TBLNEXT in B6.

The buffer parameters are initialized. Some CM words are cleared.

Also, the common's list pointer and the program's list pointer are adjusted according to the CM Loader pointer (TBLNEXT); it is necessary to do that in overlay generation or segmentation mode.

12.7.7.19 REQUEST

7.19.1 Task Description

To initiate physical loading requested by CP LOADER itself (library loading, SEGZERO loading, overlay generation or user call mode).

7.19.2 Environment Description

This subroutine is entered by return jump. Upon entry, the address of the request is in X1. The request has the following structure:

<table>
<tr><td colspan="3">File name, if any</td><td>SL list address if any</td></tr>
<tr><td></td><td>U</td><td></td><td>FWA</td></tr>
</table>

U = 1 if unsatisfied external call

FWA = First word address where to begin load.

7.19.3 Subroutine Structure

First QINITI is executed for initialization.

If FWA is not equal to zero, this value replaces CORNEXT.

If U = 1, UECALPR is entered.
If U = 0, and Lfn = 0, we are in overlay generation mode and exit is to FNCAL4, file name has been saved in OVLINFN.

Otherwise exit is to FNCAL after process of SL list by SLPROC routine if there is one.

7.19.4 Other Subroutines Referenced

QINITI	Initialization
QSSCOTB	To store CORNEXT TBLNEXT
RSETBUF	To reset buffer parameters
SLPROC	To process SL list.

12.7.7.20 PUTREQ

This subroutine puts into RA + 1 the PP request which is in X1 upon entry.

12.7.7.21 SENDRCL

Self explanatory.

12.7.7.22 SCIO

This routine calls CIO.

Upon entry, request function code is in X6.

Exit is done when the request is completed (file not busy).

12.7.7.23 NORREAD

7.23.1 Task Description

Fill up the circular buffer by using CIO.

7.23.2 Environment Description

Upon entry, return address is in B2.

7.23.3 Subroutine Structure

If the circular buffer is not empty, we return immediately after setting limit of OUT.

If the buffer is empty, the status of the file is looked at, exit is to EOF READ if status is EOF. If not EOF, CIO is called to fill the buffer and if the buffer is always empty after the completion of the request, exit is to EOFREAD, if not we return (ENDREAD).

7.23.4 Other Subroutines Referenced

SCIO	To call CIO
SETOUTL	To set limit of OUT
CHKPAR	To check parity bit in the reply.

12.7.7.24 HSPREAD

7.24.1 Task Description

To control high speed function in progress and to call LDQ for high speed read function if necessary.

7.24.2 Environment Description

Upon entry, return address is in B2.

7.24.3 Subroutine Structure

First, the subroutine looks at the status; if the file is busy, a high speed read function is in progress so we wait a little more than one sector read time. After that, if the buffer is always empty, probably we have lost one disk revolution, we send an RCL request; if the buffer is full, we return.

If the file is not busy and the buffer is not empty return is executed.

Exit is to EOFREAD if the buffer is empty and the file status is EOF.

Otherwise, LDQ is called to initiate a high speed read. We wait until RA+1 is cleared and then we return if the buffer is not empty. If the buffer is always empty, we wait the end of the request. There is no timing problem because LDQ program requests MTR to recall CP only if one sector has been read.

If the buffer is empty, there are two possibilities:

- The reply LDQ byte (6th word of FET) indicates that we have to load a library program, in this case the program name is put in a list ended by zero and FILLREQ subroutine is executed to load this routine, then exit is to CONTROL unless we are in user - call mode, in this case a non fatal bit is set in the error reply and we return to the user's program.
- The device of the file is non allocatable, in this case exit is to FNCAL4.

Whatever, the read function which is used the process of EOF status is the same (EOFREAD): there are three ways to exit:

a. in absolute overlay mode exit is to POVLT4 to initiate the execution of the overlay if it is required or to complete the load of the file.

b. in case of search file exit is to SKTB10 if there are still programs to be loaded.

c. in the other cases exit is to CMJUMP after setting end of load bit, achieve the physical loading process.

7.24.4 Other Subroutines Referenced

SENDRCL	To send an RCL request
CHKPAR	To check parity
PUTREQ	To call LDQ
WAITFF	To wait the end of request
RSETBUF	To reset the buffer parameters.

12.7.7.25 LIBREAD

7.25.1 Task Description

This subroutine controls the library read in progress.

7.25.2 Environment Description

Upon entry, return address is in B2.

7.25.3 Subroutine Structure

If the buffer is not empty, we check the error reply of LDQ, we set the new limit of out and we return.

If the buffer is empty and the switch for library read completed is set, exit is to CMJUMP.

Otherwise we wait one sector read time, and send RCL request until the buffer is not empty or the switch end of library read is set.

When the buffer is full, we return after setting new limit of OUT.

7.25.4 Other Routines Referenced

ERRLDQ	To check LDQ reply
SENDRCL	
SETOUTL	To set limit of OUT

12.7.7.26 REWIND

12.7.7.27 BACKSPL

12.7.7.28 SKIPFB

12.7.7.29 RESETRD

12.7.7.30 RESFST

12.7.7.31 QSSCOTB

12.7.7.32 RSETBUF

(12.7.7.27 – 12.7.7.31: self explanatory)

12.7.7.33 QSGOW

7.33.1 Task Description

To provide one word in X2.

7.33.2 Environment Description

Upon entry OUT is in B5, upon return the CM word is available in X2, in any case.

7.33.3 Subroutine Structure

First, if the buffer is empty, the read subroutine in usage (NORREAD, HSPREAD or LIBREAD) is called to fill the buffer. Then buffer parameters are updated and limit of OUT is set to the new value. Normal return is done with one word in X2, except in absolute overlay mode, if EOR or EOF; in this case, exit is to ROVL2 if EOR or to POVLT4 if EOF.

12.7.7.34 QSINCPL

12.7.7.35 CHKPAR

(12.7.7.34 and 12.7.7.35:) self explanatory

12.7.7.36 ERRLDQ

This subroutine fetches the LDQ reply byte (6th word of the FET) if it is less than 10B,there is no error, just some flags if it is not null.

If this byte is greater than 7 this value is the number of the error detected by LDQ, then LOADERE overlay will be entered to process this error.

12.7.7.37 SEAPIDL

7.37.1 Task Description

This subroutine looks at the programs list to determine if a specified program is already loaded.

7.37.2 Environment Description

Upon entry the program name is in X0.

Upon exit, B2 is equal to zero on match and B2 is not equal to zero if no match.

12.7.7.38 SETOUTL

7.39.1 Task Description

To load an overlay of CPLOADER (LOADERS, LOADERE, LOADERV or MAPOUT).

7.39.2 Environment Description

Upon entry first word address where to load is in B2 and last word address in B3. The name of the overlay is in X1 left justified.

Upon exit, B2 is equal to zero if the program is not read in its entirety, due to the lack of room.

7.39.3 Other Subroutines Referenced

PUTREQ	To call LDQ
ERRLDQ1	To check the LDQ reply.

12.7.7.40 RLOVLOD

7.40.1 Task Description

To relocate an overlay of CP LOADER.

7.40.2 Environment Description

Upon entry, the program to be relocated is stored in CM from SOUT to SIN; FWA where to put relocated program is in B2.

7.40.3 Subroutine Structure

First, TBLSET routine is executed to set up the relocation bases table. Then the relocation loop is entered (RLOVLOP). Each TEXT table is relocated by executing the same stock loop as the one of QPEXTT routine; all other tables are skipped.

12.7.7.41 MAP

7.41.1 Task Description

To load the overlay MAPOUT and to initiate its execution for mapping.

7.41.2 Subroutine Structure

The overlay MAPOUT is loaded in the biggest area: from CORNEXT to TBLNEXT, or in the circular buffer. Then it is relocated into the circular buffer, and entered to its first word address. If there is not enough room, a non fatal diagnostic is given and the loading process is resumed.

7.41.3 Other Routines Referenced

LOOVLO, RLOVLOD.

12.7.7.42 TBLSET

7.42.1 Task Description

To set up the relocation bases tables.

7.42.2 Subroutine Structure

Upon entry the relocation base is stored in QRELBAS. This table includes all relocation bases (positive and negative) for each position of the addresses in a CM word.

12.7.7.43 ROVL

7.43.1 Task Description

To load into the user's field length an aboslute overlay.

7.43.2 Subroutine Structure

Upon entry, overlay header must be in X5. This overlay is loaded starting from FWA as indicated in the header or from FWA user if there is one.

12.7.7.44 SETIN - self explanatory.

12.7.7.45 MOVECM

7.45.1 Task Description

To move a part of the user's area in order to load an overlay of loader or to set up the programs list.

7.45.2 Environment Description

Upon entry FWA and LWA of the area to be moved are in B6 and B2. The size of the moving is in X1.

7.45.3 Subroutine Structure

First, the area is moved. Then all pointers which must be changed due to this move, are updated.

12.7.7.46 ROOMCM

12.7.7.47 WAITFF

12.7.7.48 FILLREQ

self explanatory

7.48.1 Task Description

To load a list of library programs.

7.48.2 Environment Description

Upon entry the list of the programs name is set up in LIPRLI and B6 points to the last name of the list.

7.48.3 Subroutine Structure

First, the list is ended by a zero word and a short FET is set up, using the space between CORNEXT and TBLNEXT as a circular buffer. Then LDQ is called to read the Entry Points Table and the External Reference Table of the directory. If we have not enough room in CM, a fatal error message is issued. The Entry point table is always CM resident; the External Reference Table can be CM or disk resident.

Then, each entry point in the list of the library programs is replaced with the program number, read in the entry point table; then this list is shrunk so that all duplicated program numbers are removed.

The External Reference Table is processed only in normal load mode if this table is present in the directory; otherwise, the library loading needs several iterations.

The E.R.T. process is the following:

each program number in the list gives entry to the E.R.T.; consequently for each program number we get the list of the other program number which are referenced as external. The programs numbers which are not yet in the list, are set at the end of the list. We repeat this process until we get the end of the list.

After the list of programs numbers is set up into the circular buffer, this list is shrunk (4 numbers per word) in the table PNTBL and LDQ is called to load these library programs.

12.7.7.50 ERROR

7.50.1 Task Description

To process any error (fatal or nonfatal) detected by any module of CP Loader.

7.50.2 Environment Description

Upon entry B1 equal zero if there is fatal error.

In case of non fatal error the address of the message must be in X6.

In case of fatal error the error number must be in X6, user call flag must be in LOADERE+2, LOADERE+3 is null or contains the address of the card to be displayed, LOADERE+4 contains the name of the program which is being processed.

7.50.3 Subroutine Structure

If this is non fatal error, MSG is called in order to write in the dayfile the specified message.

Then return is done after the message is processed (automatic recall).

If this is a fatal error, the module LOADERE is loaded starting from BEGING or 100_8 if there is not enough room (in that case LOADERE will never return to user's program), and this module is entered by a jump.

Checking is made in order to detect a fatal error occurring during a previous fatal error process.

7.50.4 Error Diagnostics

The list of the fatal diagnostic is given with the explanation of LOADERE module.

However one fatal diagnostic is written by LOADERQ:

SYSTEM ERROR IN ERROR PROCESS

Usually this is a system error in CP Loader. However this diagnostic can appear if a parity error is detected during the reading of LOADERE, also if the LOADERE module is not in the library. Job is aborted after this message is sent.

Non fatal diagnostics:

- WARNING BLANK COMMON GREATER THAN PREVIOUS DECL

  If blank common has been previously established in a set of programs (SEGMENT, OVERLAY or a file loaded as the result of a control card) all subsequent references must not be greater than that which has been allocated. This is a warning only and does not abort the job, however none of the references to blank common are truncated, so it is quite possible for a program to destroy itself, if the caution is not needed.

- BLANK COMMON EXCEEDS AVAILABLE CORE, TRUNCATED

  During blank common allocation, no length will be established which would exceed FL or overlap the 20 words residence set aside for the LOADER. No reference is truncated just the allocation for core map purposes.

- SOURCE IS ZERO REPL

  If no address is furnished for the source stream to be fetched from, this warning is flagged. The source stream is than set to location RA+1 (which usually contains zero) and replication initiated.

- REPLICATION EXCEEDS AVAILABLE CORE

  During the replication process each putaway is checked to see that it does not destroy the LOADER or its tables. If this should occur, the replication table processing is terminated but loading and execution are continued.

- MULTIPLY DEFINED ROUTINE XXXXXXX

  Routine XXXXXXX has been already loaded, CP Loader gives this warning and skips it.

- XXXXXXX
  TRUNCATED LABELLED COMMON BLOCK CCCCCC

  The common block CCCCCC is declared in the routine XXXXXXX with a length greater than previous declaration. The previous length is kept, all subsequent references must not be greater than this previous length.

- FIELD LENGTH TOO SMALL FOR MAPPING

  There is not enough room to execute mapping function. The requested map is not put out and the loading process goes on as usual.

  10000_8 CM words should be sufficient to permit mapping.

- LEVELS NOT PERMITTED IN STANDARD CALL

  If the s and v bits are not on in the user call, it is interpreted as a "normal load". If l1 and l2 are non-zero, it is possible that the call was intended as an OVERLAY or SEGMENT call but the appropriate bit is missing. This warning is put out, and processing continues as if a "normal" load.

- SEGMENT CALL, BUT NO SEGZERO PRESENT

  This is the result of a user call for a segment load. If it is accomplished from a NORMAL or OVERLAY program, it is impossible to establish the necessary delinking point, since certain pointers are established for all segment loading during the initial loading of segzero. This is probably the result of an erroneous setting of the s bit.

- SL LIST EMPTY, FATAL ERROR

  This is the result of a user call containing an SL pointer which gives an address at which there is a zero word (signifying a vacuous selective load). It would be unusual for a programmer to incur the overhead for such a call when there are cleaner ways, thus this condition is assumed to be an error, and a fatal error bit is returned in the user reply.

- REQUESTED SEGMENT INCOMPLETE

  This comment results from a user call. Please note, that this message does not result from an inability to satisfy all externals, but instead arises when all the programs explicitly requested cannot be located.

12.7.7.51 SKPLIB

7.51.1 Task Description

To skip one library program.

7.51.2 Environment Description

B4 contains the number of CM words which are to be skipped on the table being processed.

7.51.3 Subroutine Structure

All tables are skipped in the buffer table by table and word by word, until a PIDL table is detected. Exit is to QPTHST to process the new program.

12.7.8.0 LOADERS

12.7.8.0.1 Task Description

LOADERS is a module of CPLOADER which is used in segmentation mode.

When necessary, it is loaded into the user's field length between the loader tables and the circular buffer (see 4.1).

Its task is to process the segment loading of SEGZERO segments as a result of a loader control card and also to process the segment loading of any segment and section as a result of a user call with segmentation.

All sequences and subroutines are exactly the same as the standard GPSL ones.

The only changes are the two new subroutines: PUTREQ, CHECKLOOP.

12.7.8.0.2 Organization

LOADERS can be entered from LOADERQ at the following points:

SEGCALL	to load SEGZERO as a result of a program call card
SEGMENT	to load SEGZERO
SEGEND	to process all loader tables as a result of NOGO card in segmentation mode
USERSEG	to load the sections or segments as required by the user

Upon entry, B1 register contains a switch to the appropriate entry point.

Returns to LOADERQ are at the following points:

USRFATL	in case of fatal error
OVLDONF	in control card mode without execution
CONT	in control card mode
SKPFLL1	in user call mode

The checksumming which is made for LOADERQ, does not check LOADERS; also a checksumming has been implemented to check the validity of LOADERS in segmentation mode. The checksum of LOADERS is named CHKLODS.

Before entering to LOADERS, LOADERQ computes and compares the checksum CHKLODS if LOADERS is already loaded.

Before returning to LOADERQ, LOADERS computes the new checksum.

12.7.8.0.3 Subroutines

There are only two new subroutines:

PUTREQ
This subroutine puts in RA+1 the PP request which is in X1 upon entry.

CHKLOOP
This subroutine builds into CHKLODS the checksum of LOADERS.

See GPSL maintenance documentation for the other routines of LOADERS.

12.7.9.0 LOADERV

12.7.9.0.1 Task Description

LOADERV is a module of CPLOADER which is used in case of overlay generation.

When necessary, it is loaded in the user's field length between the loader tables and the circular buffer (see 4.1).

Its task is to complete the overlay loading and to write out the generated overlays when it is necessary.

12.7.9.0.2 Organization

All sequences and routines are exactly the same as the GPSL ones, except the following:

An OPEN function has been added on the new file, in case of overlays on different files.

LOADERV can be entered from LOADERQ at the following points:

OFXCALL	in case of overlays on different files
OVERLAY	in case of load card with overlays
OVLCALL	in case of program call card with overlays
SETOLFG	in case of overlays on different files
OVLNOGO	in case of NOGO card in overlays generation

Upon entry, B1 contains a switch to the appropriate entry point.

Returns to LOADERQ are at the following point:

OVLDONF

12.7.9.0.3 Subroutines

There is only one new subroutine: PUTREQ. Furthermore, routine OFXCALL has been modified.

PUTREQ	This routine sets into RA+1 the PP request which is in X1 upon entry.
OFXCALL	This routine is entered only when the overlays are read on different files (several loader control cards). In this case, before loading of the overlays, the new file is opened and rewound by calling OPE and CIO.

All the other routines and subroutines of this module are exactly the same as the GPSL ones.

See GPSL maintenance documentation of these other routines.

12.7.10.0 LOADERE

12.7.10.0.1 Task Description

LOADERE is a module of CPLOADER which is called in case of fatal error detection by PP program LDQ or by any other CP module of CPLOADER.

LOADERE sends the appropriate diagnostics to the dayfile by calling MSG and the job is aborted in control card mode, in some cases return is made to the user's program in user call mode.

12.7.10.0.2 Organization

When required, this module is loaded into the user's field length, starting from the FIRST address of the circular buffer. Then it is entered by a jump to its first address.

Upon entry some parameters are set in the registers:

- The message number is set in X1.
- LOADER entry word of LOADERQ is set in X2 as a control card/ user call flag.
- X3 is null or contains the address of a card image, the first twenty characters of which will be sent to the dayfile.
- X4 is null or contains the name of the program which is being processed.
- X5 is the library loading flag.
- B7 contains, in user call mode, the return address to LOADERQ.
- B6 contains the FET address.

A message number is assigned to each type of error.

There are two sets of diagnostics:

a. Message number less than 30B
   This class includes mainly the fatal diagnostics detected by a CP module of CPLOADER, during the loader tables processing; that includes also some system errors detected by the PP program LDQ. In this case, a single fatal diagnostic is written in the dayfile and the job is always aborted.

b. Message numbers greater than 30B
   This class includes the fatal diagnostics which are detected by LOADERQ during the relocation of the text and the physical loading of a file; it includes also the fatal diagnostics which are detected by LDQ, during the loader directives processing.

In this case, several messages are sent to the dayfile.

- First the following message:

  XXXX CALL LOADER FATAL ERROR

  Where XXXX is USER or CARD

- Then the message LIBRARY LOADING is put out if the fatal error has happened during the loading of the library.

- Third

  LAST PROGRAM NAME WHICH AS BEEN READ XXXXXXX

  Where XXXXXXX is the name of the last program which has been processed by CPLOADER.

  When this message is not written, that means the error is happened before any program has been read from the user's file or from the library.

- Fourth

  The appropriate diagnostic.

- Fifth

  In some cases, mainly when the error has been detected in a loader directive, the first twenty characters of the card which are responsible for the error, are written in the dayfile.

  If one of these characters is not in display code, the first then characters are converted in octal and written in the dayfile.

Job is aborted or return is made to the user exception case of Field Length too small.

12.7.10.0.3 Subroutines

ERROR	To send a message to the dayfile	10-1
PUTREQ	To set a request into RA+1	10-2
CHKDIS	To check the characters in a word if they are in display code	10-3

12.7.10.1 ERROR

This subroutine sends the message, the address of which is set in X6 upon entry.

MSG is called with automatic recall.

Return is made after the message is sent.

Other routine referenced: PUTREQ

12.7.10.2 PUTREQ

This routine sets into RA+1 the PP request which is in X1 upon entry, return is made when RA+1 is cleared.

12.7.10.3 CHKDIS

This routine checks if all the characters in a word are in display code. Word to be checked is in X2 upon entry.

Exit: X6 = 0 if one character at least is not in display code

X6 ≠ 0 all characters are display, the word is in X6.

12.7.10.4 CP LOADER FATAL ERROR MESSAGES

12.7.10.4.1 Message Numbers less than 30B

1. OVERLAY CALL FROM RELOCATABLE
   A relocatable program has made a user call for an OVERLAY load. This is one of the new redundancy checks made to validate user call. This mode of operation is illegal, since once OVERLAY loading is initiated, control is taken away from the user call and returned to the called overlay. It is possible to accomplish the same activity, if the file named in the user call contains all absolute overlays. This will trigger, absolute mode, and if OVERLAY 0,0 has been called for, control will return to the user call.

2. LOADER TABLES GARBAGE, OR OVERLAY SEQ ERROR
   LOADER tables are threaded in a list below the LOADER in CM. If these tables are destroyed by a user program (in normal mode only), the THREAD routine will be unable to cope with the situation, the above message will be output, and the job aborted. If an attempt is made to generate a secondary overlay, without first having generated its corresponding primary overlay (or zero level) the THREAD routine will have the same difficulty, and this message will result.

4. FIELD LENGTH TOO SMALL FOR ENTRY POINT TABLE

   In order to load the library programs which are needed, CP LOADER calls LDQ to read into the user's field length, Entry Point Table and External Reference Table.

   If there is just enough room for Entry Point Table, the library loading is complete without the ERT usage, in this case there is only a warning , which is written on the second line of the MAP: ERT HAS NOT BEEN USED.

   If there is not enough room to read Entry Point Table, CP LOADER cannot complete the library loading, the above message is sent to the dayfile and the job is aborted.

5. SYSTEM ERROR IN LOADER -- HELP -- CALL CDC

   This is explanatory. LOADER has been designed to "FAIL SAFE", that is all communications with the system are checked out and edited in some manner. If one of the interfaces degenerates or a new "bug" appears in LOADER, the communication checking can result in this error comment.

6. NO TRANSFER ADDRESS

   In case of program call card or EXECUTE card, CP Loader has to give control to the user's program when the loading is completed; if CP Loader does not find any XFER table within the programs which have been loaded, CP Loader cannot give control, the above message is written into the dayfile and the job is aborted.

7. SEGZERO INCOMPLETE, JOB ABORTED

   If all of the programs specified on the SEGZERO card, cannot be found on the input file, this error message results and the job is aborted. Please note the distinction between the necessity for finding all programs explicitly called for, and the possibility that not all external references from these programs can be satisfied. In the latter case no error comment is made, since this is a normal condition for a segmented job.

10. SEGMENT CALL NOT IN SEGMENT MODE

   This is the result of a user call for segment load, when it is accomplished from a NORMAL program. This is probably the result of an erroneous setting of the s bit.

11. FIELD LENGTH NOT SUFFICIENT FOR OVERLAY GENERATION

This is self explanatory. During OVERLAY generation room must be allocated from the main LOADER of about 5400_8 locations and all loader tables for a given core map. During OVERLAY execution a great deal less core is required.

12. BAD OVERLAY FILE STRUCTURE

This error is detected by LOADERQ in overlay generation mode when the format of the overlay file is bad; specifically there are two consecutive overlay loader directives.

13. OVERLAY GENERATION AND SEGMENTATION IN THE SAME LOAD

Loading of segments and overlays generation is mutually exclusive. This indicates that LOADERQ has to load the CP module for overlays (or segmentation) and the other module is already loaded.

12. NO ENTRY MATCHES XFER

When CP loader has found the entry point name to give it control, all available entries in the LOADER table are searched. If one matching entry cannot be found, this message is put out and the job is aborted, if this was the result of an EXECUTE card or PROGRAM CALL card. In user call mode with segmentation, the non-fatal error bit is returned to the user.

13. PARITY ERROR, ABORT

Self explanatory, a parity error has been detected by CIO or LDQ.

14. SYSTEM ERROR, MODULE OF LOADER NOT FOUND

LOADERQ calls LDQ to load a module of CP loader: LOADERV, LOADERS, LOADERE, MAPOUT. If the requested module is not found in the directory, the above diagnostic is put out and the job is aborted; either one of these modules is not in the library or there is a bug in CP loader.

15. SYSTEM ERROR IN LOADER, BAD FST

In some cases, LOADERQ calls LDQ to reset the FST of the file being loaded. Before resetting the FST, LDQ checks the new FST and if it is bad, the above message is written. Normally this is an error of CP loader.

12.7.10.4.2 Message numbers greater than 30B

36. BAD BINARY DECK

This message is produced by LOADERQ. When LOADERQ has found a bad PIDL table (too many common blocks) or a bad REPL table, the above message is put out.

41. CANNOT FIND FILE NAME

This error is detected by LDQ when the file to be loaded is not found in the FNT and the name is not in the directory.

This message is followed by the illegal name.

42. FIELD LENGTH TOO SMALL

This error message is produced when the field length for the user's program is too small (the storage available between the starting point in the user area and the highest available loaction below the CP loader produced tables will not accomodate the user's program).

When this error is detected, the image being processed will not be placed in the dayfile (the image being processed is most likely a binary card, hence the display code representation of it would be meaningless).

43. BAD TEXT

This error message is produced when LDR determines that an illegal TEXT table entry is being processed: specifically, the relocation code is illegal, or the length of the table is not correct.

When this error is detected, no image is produced following the error message.

44. FILE INITIALLY POSITIONED WRONG

This error message is produced when LOADERQ determines that an input file is initially positioned at an end-of-file mark.

When this error is detected, the image being processed (EOF) will not be sent to the dayfile.

The remaining error messages will cause the image processed to be sent to the dayfile.

45. FIELD GREATER THAN 80 CHARACTERS

LOADERQ detects this error when a loader directive is improperly implemented.

Each time LOADERQ does not recognize a binary table, it tries to process it as a loader directive, even if this is not a loader directive, so the above message (or the message number 54) can appear.

46. ONLY ONE PARAMETER

This error message is produced when LDQ finds an overlay loader directive with only one level-parameter.

47. INVALID CARD FORMAT

LDQ produces this error comment, when it finds a termination character prematurely implemented on a loader directive.

50. INVALID LOADER DIRECTIVE

This error message is produced when LDQ determines that the first 7 characters on a loader directive card do not match one of the following 7 character words: SEGZERO, SECTION, SEGMENT, OVERLAY.

51. SEG OR OVERLAY CARD PREV PROCESSED

A SECTION card cannot be used in the overlay mode. When it is used in the segmentation mode it must precede all segment cards.

When LDQ determines that the rules in the above paragraph have not been followed, the message is sent to the dayfile.

52. SEGZERO HAS NOT BEEN PROCESSED

LDQ produces this error message in two instances: when a SEGMENT card is being processed and the required initial (SEGZERO) segment card has not yet been processed and, when a SEGZERO card is being processed and an OVERLAY card has been processed (segmenation and overlay modes may not be mixed).

53. SEGZERO SEGMENT NAMES DIFFER

When there are too many parameters to fit on one SEGZERO loader directive card and additional SEGZERO card may be used. This card must define the segment with the same name. When the segment names differ for contiguous SEGZERO cards, error message 53 is sent to the dayfile. This is determined by LDQ.

54. NAME GREATER THAN 7 CHARACTERS

When LDQ determines that a name used on a loader directive card is greater than 7 characters in length, this error message will be produced (see message number 45).

55. NO TERMINATOR FOUND

When LDQ determines that a loader directive card does not have a legal terminator . or), this error message is sent to the dayfile.

56. INVALID CHARACTER

This message is produced when LDQ finds a character on a loader directive card that is illegal. The legal characters are letters, numbers, parentheses, blanks, commas, and the period.

57. SEGMENT OR SECTION CARD PROCESSED

When LDQ is processing in the overlay mode and determines that a SEGMENT or SECTION card has been processed, this error message is produced.

60. 1st OVERLAY CARD HAS NO FILE NAME

When LDQ determines that the first character of this first parameter on the intial overlay card is not alphabetic, this error comment is produced.

61. 1st OVERLAY CARD LACKS 0,0

When the option to load the overlay a designated number of words above blank COMMON is used on a level zero overlay card, this error message is produced. LDQ detects this error.

63. 1st PARAMETER MAY NOT EQUAL ZERO

The zero level overlay may not have secondary overlay levels, e.g., 0,1 is illegal. LDQ detects this error.

64. ONLY 1 OVERLAY DESIGNATOR USED

When LDQ determines that the user has not designated both a primary and secondary level on an OVERLAY card, this error message is produced.

65. C OPTION NOT LAST PARAMETER

When LDQ determines that a termination character does not follow the c option on an OVERLAY card, this error message is produced.

66. TOO MANY CHARACTERS IN PARAMETER

When LDQ determines that a level number on an OVERLAY card is greater than 77, this error message is produced.

67. C OPTIONS DOES NOT START WITH C

The option used on an overlay card which allows the user or designate how many words above blank COMMON the overlay should be loaded must have the alphabetic character "c" as the first character.

When LDQ determines that the 1st character is not "c" this error message is produced.

70. DIGIT IS NOT OCTAL

    When LDQ determines that a digit used on an OVERLAY card is not octal, this error message is produced.

71. TEXT HAS BEEN PROCESSED

    When LDQ is processing a SECTION card and determines that text has been previously processed, this error message will be produced.

73. ERROR IN ABS. OVERLAY FILE FORMAT

    This error message is produced by LOADERQ when LOADERQ is in absolute overlay mode and determines that the identification code of the subroutine being processed is unequal to 50.

12.7.11.0 MAPOUT

12.7.11.0.1 Task Description

MAPOUT is the module of CP loader which executes all mapping functions.

According to the user's options, this module can write out

- a full map — that is the addresses of the programs and common blocks; also the addresses of all entry points with cross references and then the unsatisfied externals.
- a partial map — that is only the addresses of the programs and common blocks, also the unsatisfied externals.

This option is set by control card: MAP(PART)

- in user call mode, when partial map bit is set, a partial core map is given.

12.7.11.0.2 Organization

There are three MAP options which can be set by control card:

MAP(OFF)	No map in control card mode
MAP(ON)	Full map in control card mode
MAP(PART)	Partial map in control card mode and user call mode.

One of these options is chosen as a default option.

A mapping function will be executed in the following cases only:

- In control card mode: if at least one user's program (a program not in library) has been loaded and if the option MAP(OFF) has not been set, partial MAP will be furnished if the option MAP(PART) has been selected, otherwise a full map is given.
- In user call mode: if the no-map flag (bit m) is set to 0 in the user's call, whatever the option MAP OFF or MAP ON. A partial MAP will be written out if the option MAP PART has been selected by control card (or installation option).

When a mapping function is required, MAPOUT module is loaded by LOADERQ, into the circular buffer, starting from the FIRST address. It is entered by a return jump from LOADERQ to the first location of this module. Upon entry a user call flag (LOADER) is set in X7 and an external reference table usage flag is set in X6.

The code of this module is exactly the same as the GPSL one except the following points:

- automatic recall is always used to write out the MAP

- before writing the core map, this module chooses the location of the circular buffers as the following:

  The circular buffer is located into the biggest area: into the user's field length, from CORNEXT to TBLNEXT or into the rest of the circular buffer.

  If the length of this buffer is less than one disk sector, a non fatal diagnostic is put out:

  FIELD LENGTH TOO SMALL FOR MAPPING

  and normal exit is taken.

- An information has been added to the second line of the MAP, that is the following message:

  ERT HAS NOT BEEN USED

  In the following cases, the External Reference Table is never used to load the library programs:

  - User Call
  - Segmentation
  - Overlay generation

In these cases the above message is never written.

Otherwise (normal loading in control card mode), ERT can be used; in this case, if ERT does not exist in the directory or the user's field length is not sufficient to read it, library loading is completed with out ERT and a flag is set in order to write the above message on the second line of the map.

5.0 - HOW TO LIVE WITH THE SCOPE 2.0 LOADER

In the course of testing LOADER and putting it into use with major programs such as RUN, ASCENT, PERT, and APT, a number of interesting facets of its flexibility were revealed. The following discussion is based on these discoveries, and is intended to aid programmers in the use of LOADER.

5.1 GENERAL PHILOSOPHIES OF IMPLEMENTATION

The LOADER was implemented in the manner described previously to remain within certain desirable constraints:

A. Minimization of number of control cards.

B. Provide a basic similarity between system usage of CHIPPEWA 1.1 and SCOPE 2.0, for normal jobs.

C. Accomplish as much physical loading of material without the use of the CP, leaving the central processor tasks better suited to its computational power.

D. Provide the programmer with substantial flexibility.

E. Provide as much opportunity to establish an audit trail of loading and processing through dayfile warnings and core maps.

The implication of these constraints is that the LOADER has taken on certain characteristics, which are not common to most loader systems:

A. Many more options are provided by the user call than are available from control cards. In particular, the suppression of core maps and selective loading of named programs from a file can be accomplished only through user calls.

B. Many jobs can be run interchangeably under COS 1.1 and SCOPE 2.0. However, this may lead to inefficient use of the features of SCOPE 2.0 and corresponding increases in throughput time due to the overhead of the new system.

C. Total elapsed time for job execution is longer under SCOPE 2.0 than under COS 1.1, although CP utilization is less than one per cent greater. This is due to the use of a PPU for loading and relocation of program text.

D. The system, as implemented has a great variety of options which offer considerable flexibility at the expense of increased complexity and a greater need to understand the inner workings of the LOADER. When this flexibility is provided, the overall diagnostic capability of LOADER suffers, since redundancy and mutually exclusive options are reduced.

E. The inclusion of core maps and extensive error comments impacts both speed and core size requirements. Over 20 per cent of the CP LOADER care requirement, and 25 per cent of its execution overhead are due to the core map routines and error diagnostics. A special overlay had to be constructed for the PPU routine LDR to handle the diagnostics of that program.

F. Overall throughput of SCOPE 2.0 has been decreased somewhat depending on the mix of jobs. This is the result of high disk activity associated with the loading process, since not only the programs to be loaded must be retrieved from disk, but also LOADER, LOD, LDR, 2LB, 2LA, and 2LE.

5.1.1 Features Not Detailed in the ERS

The following are options of loading or implications of various usages of LOADER not explicitly spelled out in the ERS.

A. Normal Loading

A multiplicity of files may be loaded for one job and then executed. Assuming that program text has been loaded on files A, B, C, and D from the card reader or through compilation or assembly, a deck setup of:

```
LOAD (A)
LOAD (B)
LOAD (C)
LOAD (D)
EXECUTE (A)
```

would result in the load of all programs from the four files. Note that all programs will be loaded even though some of these may have the same name. The only option which excludes this option would load only programs from File D not loaded from A, B, or C if it replaced the "LOAD (D)" card in the deck.

All programs in the four files will be linked together, since their loader tables are 'added on' to the ones previously loaded. Once the system encounters an EXECUTE, NOGO or program call card, all of these previously loaded LOADER tables are cleared; thus subsequent loading is treated as a brand new load. For example:

```
LOAD (A)
EXECUTE (START)
LOAD (B)
EXECUTE (NEXT)
```

In this case, Program B will be loaded over the top of Program A, and they will not be linked to each other.

When normal relocatable programs are loaded by control cards, the following actions always occur:

1. A core map is produced.

2. All unsatisfied externals are filled in with out-of-bounds references.

3. Any external reference not satisfied, initiates an attempt to load library programs from the RSL or CLD.

4. All files from which programs are to be loaded are rewound, except the INPUT file.

5. Programs may or may not be separated by end of records when there are more than one on a file.

6. Loading proceeds to the end of file or to two consecutive end of record cards.

Implications:

1. If the user does not want to clutter up his printouts with the initial core map, he should rewind the output file prior to proceeding with program execution. This cannot be done with control cards, since the core map is produced as the result of an EXECUTE, NOGO or program call card, not the LOAD card.

2. Although a job running in NORMAL load mode may execute "user calls" for additional loading, any loading that is accomplished is subject to the following constraints:

   a. All loader tables are added on to those in core already. Linking of externals in the new programs loaded will proceed from entry points in the new programs being loaded which have "matching twins" in the old programs will never be linked to.

   b. Since all externals which remained unsatisfied at the completion of loading the old programs have been filled in with out-of-bounds references, they no longer appear unsatisfied to LOADER. When operating in normal mode, LOADER will not DELINK these out-of-bounds references. Thus, no externals from the old programs will be linked to the new programs. The only way to effect an entry from the old programs into the new is to use the ENTRY address returned by the LOADER in the reply to the user call. This address will be to the entry which appeared on the END card of the last program loaded. If it is desired that other entry points be accessed from the old programs, a user call with the "K" bit on (indicating a search) and the desired entry point name in bits 18-42 (Fn) can be executed. LOADER will search all of its tables to find the matching entry. If found, it will be returned in EA of the reply. This scheme would appear to have little utility; however, it provides the programmer with a handy barrier between programs which he wishes to reside in core, but which may contain conflicts in names, labelled common allocation, or which need to be brought in dynamically.

3. In control card mode the user cannot control the linking of his programs to library routines since a library search is always conducted. Thus, if the user wishes to test a routine which has a counterput on the library, he must load it into core first, put it and all referencing programs into a segment or user call load or ensure that its entry points are unique.

4. All programs on any file, not the INPUT file, will be loaded, since these files will be rewound. Only programs on the INPUT file from its present position to an end of file or double end of record will be loaded. The user is thus cautioned to be cognizant of the position of the INPUT file at the point where the loading is requested. Most common mistakes have been to position the INPUT file at the data record (if there is one) or omitting the double end of records. LOADER will attempt to load the material encountered, and depending on the job, some type of error comment will be produced. Since when this data is encountered during the loading process it is in display code format, it is possible for LOADER to misinterpret what is present, thus triggering unusual error comments. For example, let us say that the first data card in the deck contains a slash "/" in Column one, and that the programmer has erroneously omitted the second end of record which should divide program text from data in INPUT file. This slash "/" will appear as a 50_8 left adjusted in a word, and will be interpreted by LDR as an absolute overlay header. The appearance of this "header" triggers LDR into absolute overlay loading, but this routine detects that the file also has been loaded as a relocateable file, and since such a mixture is prohibited, the error comment "ERROR IN ABS OVERLAY FILE FORMAT-" will appear.

Miscellaneous:

1. The use of the FILL flag when making user calls in normal loading mode can be quite valuable. If the FILL flag is left zero in the call, the programs which are loaded as a result of the call will retain all unsatisfied externals with a <u>zero</u> as the absolute external address, instead of an out of bounds value. Thus subsequently loaded programs will be linked up to these references. On the other hand, if the programmer wishes to "close off" these externals, he may set the FILL flag on, causing all such references to be filled with the out of bounds quantity. Subsequent programs will not be linked up to these externals since in normal mode these external references are <u>NOT DELINKED</u>.

2. The R or reset bit in the user call is provided to permit the programmer to initiate a "brand new load" with a user call. This "brand new load," is accomplished by resetting the loader table pointers and CORENEXT so that loading commences as if no programs had been yet loaded. This means that program text will begin loading at 100_8 and tables at FWALODR-4. Such a user call <u>must</u> be established in core

outside the area which will be loaded into with either text or loader tables. The most common usage is to setup a "control routine" which is origined below 100_8 (but not in $64\text{-}67_8$), which calls for the required loading, then makes a user call for a search for a specific entry point (K bit on) and then return jumps to that entry. Ultimately control is returned to this "control" routine via the return jump "exit". Note that, since all pointers are reset before the load, no linking will take place between this "control" routine and the newly loaded material. Thus the (K search) and return jump are the major means of communication.

3. If the core space available for loading programs as a result of a user call or control card is insufficient for the loading of the specified programs, different error indications can arise. This situation arises when an LWA is specified or TABLENEXT is such that there isn't room for the complete load.starting at FWA or CORENEXT (when FWA is not specified). Three possible occasions arise:

   a) The program length on a PIDL table is greater than the available storage.

   b) In the absence of this length, TEXT table loading cannot continue because of no available storage (for the text table!).

   c) There is insufficient room to load one or more of the library routines to satisfy externals.

   In the user call mode, each of the error comments resulting above will be preceded by the words "USER ERROR". In the control card mode the first two errors above will result in comments which are preceded by "LOADER". In the third case, the CP LOADER is actually controlling the loading of library routines with "User Calls", thus the error comment will be preceded by the words "USER CALL" although this occurs in control card mode.

   In either of the two situations above the conditions causing the comments will leave core in a peculiar situation. If the error was detected in control card mode for a & b above, the job is aborted. If detected for C above or if for a, b, & c in "user call" mode, any tables already in core will be linked up, the fatal error flag set and the job continued. In this case, it is possible for sufficient code to be linked up to permit initiation of a routine which never entirely got into core, since LINK and ENTRY tables many times will precede the TEXT tables. This situation would result in the job "bombing out" in an unpredictable manner. If the PIDL table contains a length which triggered the error comments, no further loading is accomplished, and no loader tables would be loaded. In this case the "bomb out" would be due to a Mode I or Mode II error, resulting from filling externals with out of bounds references or leaving them zero.

4. Any user call to LOADER should be followed by tests of the FATAL error or NONFATAL error bits. A most common problem is the complaint that LOADER "hung up" when the CP comes to a screeching halt with a P=1. This occurs often when LOADER is unable to load the requested programs and returns to the user with the fatal error flag set, and EA equal zero. A program may then pick up EA indiscriminately, without determining that loading was completed properly; and this address is usually placed in an RJ instruction which is then executed. This, of course, will cause the "hangup" mentioned above.

5. Properly operating programs, when executed singly, may on occasion interfere with each other when linked together. The results may not be detected by LOADER with more than a warning message. The results will be unpredictable. For example, assume two programs which both utilize blank common. The first, Program A, declares blank common to be 1000_8 locations; the second, Program B, declares a blank common of 2000_8 locations. Now, if A is loaded and executed, and during this execution calls for the loading of B, by the time B is loaded, blank common will have been established immediately following A and with a length 1000_8. Program B will thus be loaded immediately following this blank common area. However, when the blank common processor is called for Program B (after all other loading has been completed), it is discovered that the two declarations of length are incompatible, and that the subsequent one is larger than the currently established length. Since blank common has already been "bounded" by the load of text from B, it cannot be reestablished. Therefore, LOADER issues a warning that the declaration of blank common in B is in error; however, all references to this blank common are left intact. If Program B makes use of the full length of this 2000_8 word blank common for storing material, it obviously will be storing data upon itself, resulting in strange and most often undecipherable results, particularly since when tested independently, the program may easily have run correctly.

6. FILL tables are processed after REPL tables, resulting in further strange results when the programmer is attempting something "tricky". For example. let us say that both Program A and B utilize labeled common X. Program A origins some code (which may never be used) into labeled common X. However, when used in concert with other programs, this code is not accessed. Let us say then that Program B utilizes this labeled common for table look up, and that these tables are produced through replication from some block of core. Since REPL tables are processed first, these tables will be established as required; however, prior to initiating execution of the program, all FILL tables for all programs are processed. This will introduce some addresses right into the middle of the replicated tables and perhaps destroy the program execution.

B. Segment Loading

All comments regarding normal loading apply to SEGMENT loading. However, The mechanism of initial loading should be restated so that the programmer may understand some of the resulting error comments. In normal loading the LOAD (A) card would initiate the loading of all programs in File A. This entire file load is accomplished by LDR before LOADER is ever called into action. Any difficulties encountered by LDR will result in dayfile error messages preceded by the words "LOADER ERROR". In segment loading, LDR loads into core only the SECTION and SEGMENT tables. LOADER is then called to formulate the first (SEGZERO) segment. LOADER accomplishes this by making "quasi user calls" to LDR for the loading of various programs. Thus, although initiated by a control card, the loading of SEGZERO can result in error comments from LDR preceded by the words "USER ERROR".

The implementation scheme used to delimit segments so that they may be linked and delinked is spatially oriented. This means that the entity "SEGMENT" is defined from the last word of the previous segment tables to the last word of its own segment tables. When segmentation is performed in a straightforward fashion, this poses no problem; but two conditions can turn this whole process into a nightmare for LOADER and the user:

1. When code or data is origined absolutely, and before the first word of the given segment.

2. Or when a particular segment is loaded using the FWA option, far out into core, and higher level segments are loaded below this segment.

The results of 1 (above) are to establish linkages with external references in lower level segments to part of the segment which is not within the boundaries of the "entity". When this "entity" is delinked, the references to these out-of-bounds items will not be delinked, and the outcome is virtually an orphaned bit of code. For example, let us say that SEGZERO contains a Program A which has external references to an ENTRY point X imbeded in it. Further, let us say that this "tricky" programmer chose to origin this code at location 50_8, and assume that he was intelligent enough to keep from wiping out the LOADER pointers at $64\text{-}67_8$. The external references in SEGZERO to ENTRY points in level 2 segment will all be filled in with addresses whose magnitude is greater than the last word of the segment tables for SEGZERO, since the next higher segment will be origined immediately following these segment tables. This is true, with the exception of references to entry point X which is floating around in low core.

Now let us assume that a user call is made for the load of a new segment at level 2. This will initiate the delinking of the old segment. This is accomplished by scanning down the LINK tables for SEGZERO and clearing out any references to addresses which are greater in magnitude than the

FWA of the old segment level 2. Since the errant code from this old level 2 segment is below that address, all references to entry point X will fail the delinking test, and they will not be cleared. Thus forever more, entry point X is inextricably linked to SEGZERO, a situation which will continue until SEGZERO has been replaced.

In the second case, the results are much the same, but the method of arriving at them is different. Let us say that in the example above, the user call for the loading of the new segment level 2, contained an FWA about 10000_8 locations beyond the end of the SEGZERO tables. This will force the segment to be loaded far out in core, instead of adjacent to SEGZERO. Let us further assume that this second level segment contains unsatisfied external references to entry point Y. Then let there be another call for the loading of a segment at level 3, with no FWA specified for the loading. LOADER philosophy dictates that this segment be loaded at CORENEXT, which points to the next available location in core, at the end of the SEGZERO tables. This means that a higher level segment will be loaded into core at a lower address than the lower level (2) segment. Thus references to entry point Y will have absolute addresses which are smaller in magnitude than the address of the last word in the segment level 2 segment tables. If a user call is then made for the loading of a new segment level 3 the delinking process will find no references to delink in the level two segment (since there was code loaded beyond it in core); however, any reference to entry point Y in SEGZERO would be delinked. If the incoming segment level 3 is forced to load beyond the level 2 (using the FWA option) and this new segment also has an entry point Y, we have the strange situation of SEGZERO being linked to the new level three, and the level 2 segment being linked to the old level 3. This will continue until the level two is delinked, at which time, of course, the old level 3 will cease to exist. There is some utility in this. However, some clever programmer will certainly find some extraordinary scheme which can effectively employ this feature.

Some confusion exists regarding the use of the FILL flag in segment mode. Under the implementation of the present LOADER, all references are delinked if their address points beyond the FWA of the segment being delinked. Since out-of-bounds references appear to be such addresses, they are cleared like any other delinked quantity. Therefore, the use of FILL does not provide a barrier to inhibit linking of certain externals in segmented mode. It serves in this instance only to provide an alternate indication of a source of error (a mode 0 appearing when the F flag is zero and a mode 1 when the F flag is one, if the instruction at fault referenced an unsatisfied external).

The present implementation of loader creates one major imcompatability in program sets run under segment mode, and under overlay or normal mode at different times. In the normal and OVERLAY modes, labeled common is linked up between all programs. This means that a block of storage is allocated for a specific labeled common when LOADER encounters the first declaration of it, and thereafter all references to that labeled common block will be adjusted by loader to reflect the actual allocated position in storage. This process is true for all programs in a given load. It is also true for all programs in a given segment. However, in segmentation

mode, linkage to labeled common is restricted to within the segment in which it is declared. Thus data communication between programs follows the rule that within segments the programmer may use either blank or labeled common, and between segments data must be communicated through blank common only. This is a severe restriction only when the data which must be communicated is preset into labeled common during assembly or compilation, and loaded there by loader. In such a case the data must be moved into common by 'object time' instructions.

Programs are not loaded into core in the order in which they appear in the user call parameter list, or section segment tables. The order is:

1. Named segments or named sections.

2. Individual programs.

Within a named segment, any named sections are loaded first. Then individual programs in the segment are loaded.

The list of individual programs to be loaded, either as a component of a segment, section, or neither is processed in the following manner by LDR. A read of the required file is accomplished starting from the current position of the file. The PIDL table name is then isolated and compared against every name in the list. If a match is found, the address of the first word of the program is loaded. The next PIDL header is fetched and the list searched again. If no match is found, LDR skips down the file to the next PIDL (note that an end of record is not necessary as a separator). This search procedure is containued until all addresses in the list have been satisfied or until the file has been searched completely back to the position from which the current search started.

LDR is present with separate lists for SECTION, SEGMENTS, and single programs. When a user call SL list contains either a section or segment name, an address is returned to the user in bits 0-17 of the corresponding entry. This address is the origin of program text (not labeled common). for the first program loaded for that section or segment.

It should be obvious from the above discussion that the addresses in the SL list returned to the user will not be in numerically sequential order, proceeding down the list. Therefore, the user is cautioned not to expect the first address in the list to contain the first word address of the "load".

Since segmentation can be effected with some ease, it is possible to come up with somewhat "sloppy" allocation of core and utilization of IO. The most common mistake is to prepare a number of programs, which might contain over 40-50K words on a single file, and to make frequent user calls for the loading of a given segment. The following are some recommendations for effective utilization of the segmentation feature:

1. Either explicitly or implicitly force the loading of all library subroutines in SEGZERO. (Explicit forcing would be accomplished by "dummying up" external references to routines which otherwise would not be called to satisfy externals for the programs actually in SEGZERO.

2. Leave the C bit off in all user calls (not needed, since we have now loaded all necessary library routines).

3. Organize routines on the file in the order in which they are going to be called.

4. Where frequent calls will be made for routines, which would make such an ordering impractical, these routines should be copied onto their own individual files, and the corresponding user call set up.

5. Make the first few runs in full core map mode to determine if programs have been allocated to given segments effectively.

6. Run many of the subsequent runs in partial map mode to determine the frequency of access and efficiency of utilization.

7. If the results of these runs indicate that segments can easily be prestructured, convert the whole shebang to OVERLAY format to speed up loading.

These steps will become quite obvious once the user has witnessed LOADER making an "end-around search" of a file of five or more programs of several thousand words. This action requires considerable disc access, and results not only in a single job slow down, but due the higher probability of disc conflicts total system throughput can be substantially affected. By identifying separate programs as files, little disc space is wasted, while the search time is eliminated. With the C bit off LOADER will make no attempts at satisfying externals, which can take a small amount of time from the job.

Blank common is allocated immediately following the first segment which declares it. Thus in many cases blank common cannot be used to overlay portions of LOADER for data areas to conserve core space, since other segments will follow the blank common area in core. This must be considered when deriving core storage requirements. Any user call can be executed if the last 20_8 words or core (FL-20) have been preserved, since these cells contain a routine to reload LOADER if necessary. If these cells have been "clobbered" during execution of the object program (LOADER can never load anything on top of itself), the program will most ungracefully hang up.

Once blank common has been allocated, it is fixed in place until the segment which first declared it is delinked. It can be effectively "shrunk down or expanded, by using the FWA option in the user call for the next level segment, which would normally be loaded immediately following blank common. Thus the user can "FWA" this segment right into the middle of a declared blank common area, if he so desires, thus contracting blank common temporarily.

Several users have been confused when examining core maps of segment jobs, since they grow to expect that their first program begins in 100_8.

Since section segment tables are stored at this point, the programmer should look for the end of the segment or section tables (a zero word sentinel), program text follows immediately.

C. Overlay Loading

During the generation phase of overlay loading, all of the comments concerning normal loading are applicable, with one exception. The programs in each overlay must be separated by end of record cards and the appropriate OVERLAY card.

OVERLAYS must appear on the incoming file in the order of their core arrangement. That is the first overlay encountered must be a 0, 0 overlay, the second a 1, 0; 2, 0; 3, 0;....n, 0 (primary), and third a secondary overlay, if there is one associated with the particular primary. All secondary overlays for a specific primary must then appear before another primary appears. At least one program in each overlay must have an END card with an entry name for a point in the overlay. If not, an invalid entry point will be established for that overlay.

The generation phase results in the production of absolute binary text of a set of programs which can load into core twenty times faster than normal relocatable text. The comments regarding assigning certain highly used overlays or segments, made in the previous section are more applicable to OVERLAY than segmentation. This is due to the obviously high rate of speed (approaching that of COS 1.1) available in this mode. It is, therefore, easy to gain a great degree of efficiency by reducing the search overhead to locate a specific overlay.

The LOADER makes no core maps during the loading of absolute overlays, but it can make core maps during the generation phase.

Upon completion of the generation phase, no end of file is written on any file. This is to permit the use of more than one input file to the generation. If the output files were written to tape this can cause a problem, since any end-around search could cause a read beyond the recorded information, and a probable parity error.

The construction of a core allocation "tree" for OVERLAYS is more difficult, because once created, it is frozen and cannot be changed at object time, as can segment processes. The cleanest "cut and try" method arrived at has been to use segmentation mode to arrive at a proper allocation of overlays, then the programmer modifies his user calls and directive cards to convert to the OVERLAY mode.

A zero-zero overlay <u>may</u> call another zero-zero overlay. This results in LDR initiating the CPU at the entry point of the new overlay rather than at the instruction following the user call, since it is assumed that the new overlay will be loaded on top of the old overlay, thus destroying the user call, and making such a return meaningless.

No writing of absolute overlays is ever done for core cells below 100_8, since this area is reserved for pointers and parameters during overlay

generation. The user must check to see that no code to be loaded is origined in this area (such as a 'tricky' control routine) for it will be lost during the generation phase.

Labeled common is linked up between OVERLAYS. An Overlay 0, 0 must be the first record encountered on the first file to be loaded, via control cards; otherwise the job will abort, if other overlays are present.

Overlays may reside in the CLD, but not the RSL. This is due to the fact that the only non-program word in an absolute overlay is the header, whose lower 18 bits (0-17) must contain the address at which the overlay is to be loaded. This usage prohibits the use of this location for threading to the next RSL entry. Such overlays are called by name, not level.

An overlay may call another overlay, normal program or segment. This is done by the RUN compiler which calls on ASCENT and which can also call for the loading of relocatable text (in RUN (G) mode).

During absolute overlay loading, any level overlay may call any other level overlay; however, if such a call is outisde the well defined tree (i. 0, 0; 1, 0; 1, 1; 1, 2;..1, n; 2,0; 2,1;..2, n;.....m, 0; m, 1; m, n) the results may be unpredictable. For example, an overlay could load on top of itself (with the exception of the 0,0 case, stated above), or a 3, 4 overlay could be called in while the primary overlay was still 1,0, perhaps 'clobbering' a portion of that 1, 0 overlay.

Each overlay is written to a file with a 77_8 ID record, containing the name of the first program in that overlay. Thus the overlay file could be updated using COPYN very readily. The process of updating absolute overlay files calls for special attention. For example, assume that there were 100 overlays in the system ranging from 0, 0 to 77, 77, and that it is desired to change the 0, 0 overlay. Most likely any change to an overlay will result in a change in effective length of the overlay (thus affecting the origin of subsequent overlays) and the relative position of entry points in the overlay will also most likely change. This means that all overlays in the system will have to be regenerated. Now assume that a single overlay at the highest level must be modified. This 77_8, 77_8 overlay will have to be regenerated from the modified relocateable decks. This regeneration will require the generation (albeit redundant) of the 0, 0 and 77, 0 overlays, to provide both an origin and necessary entry points for the generation of the 77, 77 overlay. In this latter case it is obvious that only three overlays need be generated. In both instances, although the generation involves more than one overlay, the process of updating the files need only utilize the overlay of interest, since the redundant generation should have produced overlays identical to those on the file.

RANDOM THOUGHTS TO KEEP YOU STIMULATED

A. Philosophy of Program Origin

In order to implement this relocateable system it was necessary to derive some conceptual approach to the problem of defining a "formal" origin from a program. This is due to the fact that code from any given program can be origined absolutely any place in the job area, within labeled common or relative to a program 'beginning'. In the cases where it is necessary to relocate either data, code or references relative to program origin, it is necessary to compute a displacement from the RA to that program which can then be added to the address which is relative to the program origin, to give an absolute address which will be used at execution time. When a straightforward program consists of nothing but text to be loaded starting at location 100_8 (first relocateable program), the program origin is obviously at 100_8, and that will be the displacement used in relocations for that program. Let us say that this program extendes to 777_8 and that another program is to be loaded from the file. This program (for the sake of simplicity we'll give it mythical, straightforward properties) will begin loading at 1000_8, which is obviously the program origin and the value of the displacement to be added to relative addresses.

Now lets add on labeled common to these two programs. Let the first program have a block "A" of 100_8 words and the second program have a block A and a block B, where B contains 50_8 words. LOADER will allocate 100_8 words for A before loading the first program, which will start at 200_8 this time. When the second program is loaded, block A will have already been allocated, so 50_8 words will be set aside by LOADER for "B". This block will start at $(100_8+1000_8)=1100_8$ and will continue to 1147_8. The second program will then be loaded after its labeled common block at 1150_8, and this value will be used as the program origin and displacement for relocation.

Let us say that a program consisting entirely of absolutely origined code, to be located at 50_8, is to be loaded as the second program. Since it is absolute, loader will place the code properly, however, the program origin will be considered to be 1150_8 (assuming still a labeled common of 50_8 words. This seeming incongruity arises because no distinction is made between a program containing purely absolute code, purely relocateable or purely labeled common origined. Since a value must be established for the displacement in case it is needed, the program origin is placed at the first location in core where a table containing relocateable text and a relative address of zero would be loaded of there was one.

The purpose for this discussion is to explain why in some cases the program origin on core maps does not correspond with the actual location of code. In addition the ASCENT user may wish to deal with relocated addresses for absolute, labeled common and standard relocateable code, and the above concept must be clear before attempting a sophisticated handling of such addresses.

B. Uniqueness of Program Identification

When programs are to be selectively loaded from a file, there are two definite contributors to its unique character, name and position on the file. Several copies of a program can exist on a file, and in the case of segmentation, such a procedure can increase efficiency. The rule followed by LDR is that the first program encountered which matches a name in the SL list, is the only program loaded into core. Thus the position of the file at the time the user call is issued will determine which copy of a program is loaded.

When the SL list contains more than one identical name for a program only one copy (the first copy) of that program will be loaded. In normal loading, if the program has already been loaded, it will be loaded again, but a non-fatal error flag will be returned to the user, indicating that a possible redundancy exists in the system. Since a program containing the same name need not have the same entry points, this method of loading might be useful to someone. However, if duplicate copies of a program with duplicate entry points are loaded into core, only the entry points in the first encountered program will be linked to (with the exception of a special case in segmentation).

Routines are loaded according to different identity tags depending on the method of loading. For example, if a user call contains a request for loading of a library routine, the SL will be zero and the Fn will contain the ENTRY point name required. The CLD and RSL are, therefore, searched only on entry point names. If a selective file load is indicated, the SL list will contain program names which appear on the PIDL card, and which may easily differ from the names of any entry points contained therein.

C. On the RSS Bit

A convenient technique for debigging under 2.0 is to execute a load and then a NOGO. This permits the programmer to DIS to the appropriate control point for debigging purposes. When the programmer attempts to use the RSS option as he did under COS1.1, he will note a slight difference in behavior of the system. This is due to the fact that if the RSS bit is set, all loading is completed, and a core map output, then LOADER places an END in RA+1 and waits to be dropped. This is the same as the NOGO behavior. Unlike COS 1.1, the loading of core requires some CPU activity; thus this activity is completed prior to stopping before CPU execution of the loaded job. This can cause the programmer some difficulty, for RSS only works to stop the execution of CPU routines, and does not inhibit the operation of any PPU routines which may be called up.

D. Another source of some consternation has been the fact that LOADER is most tolerantly intolerant of the data it is loading. This means that as much editing as possible is performed on the input stream to ensure its validity. Thus if the binary card deck were 'shuffled' out of sequence, LRD can encounter binary stream information as well as display coded "loader

directives," it is apparent that if given the wrong file position, loader can make a wrong 'guess' and diverge on some ridiculous wild goose chase, trying to load something it shouldn't.

If an error is detected in the input stream while loading in the control card mode, LOADER will issue the best error comment it can muster, and the job will be aborted. Should the error result from a user call, the job will not be aborted, but a fatal error flag will be returned to the user. In either case, the file will be positioned at the physical disc record immediately following the error. LOADER does not attempt to recover the loading procedure and continue, since it is virtually lost in the input stream, and since the user is not restricted to separating most of his programs with end of record cards.

E. Things to Look for When the Loader Has Apparently Gone to Hell

The infrequent user of SCOPE must suffer the problems of orienting himself to the relocation of an otherwise simple program and the increased complexity of debugging something which "ain't where it's supposed to be at". Some typical things which have happened to typical users have resulted in the following observations:

1. A nice normal looking load turns to chaos with the error comment: "No terminator found." The user will probably find that he forgot to remove the blank card supplied by the COS output package. LOADER is attempting to scan an apparent "loader directive".

2. Program hangs up in high core--user has probably stored data on top of the loader residence.

3. Program hangs up in low core--user has failed to test error flags in reply and has tried to return jump to nowhere, supplied by the EA of the user call.

4. Program initiation seems faulty--either the user failed to heed the warning "NO ENTRY FOUND FOR XFER" or his END card specifies the wrong ENTRY POINT or more than one END card is extant. (Loader takes the last one loaded.)

5. Strange activity in normally running program, data, or other garbage loaded as program text--a file name exists for the job which matches a routine which is being called from the library. (The order of search is FNT, RSL, CLD, RPL, PLD.)

6. INVALID CONTROL CARD appears indecipherable--in addition to editing the card contents, LOD outputs this message when a program call card appears with a name which is not in the FNT, CLD, RSL, RPL, PLD.

7. An invalid card message followed by garbage on the dayfile--commonly arises when the file is positioned wrong and LDR tries to interpret the binary text at that point as a display code loader directive.

8. If all other analysis fails, then 'immediate action' is:

- Look at RA+64; contains file name of current file being loaded.
- Look in RA+65; contains address +1 of last location where physical loading took place; RA+67 contains TABLEXT which points to last set of tables loaded.
- Find the last loaded PIDL; check to see if that entire program was loaded (FWA) is in PIDL, LWA+1 in CORENEXT.
- If in segment mode check section segment tables at RA+100_8.

The above information should indicate in which routine everything died. Examine that deck carefully for card sequence, blank, or other spurious cards. If there is data on the same file, check to see that there is a double end of record between the programs and the data.

FIRST 100_8 LOCATIONS OF PROGRAM AREA

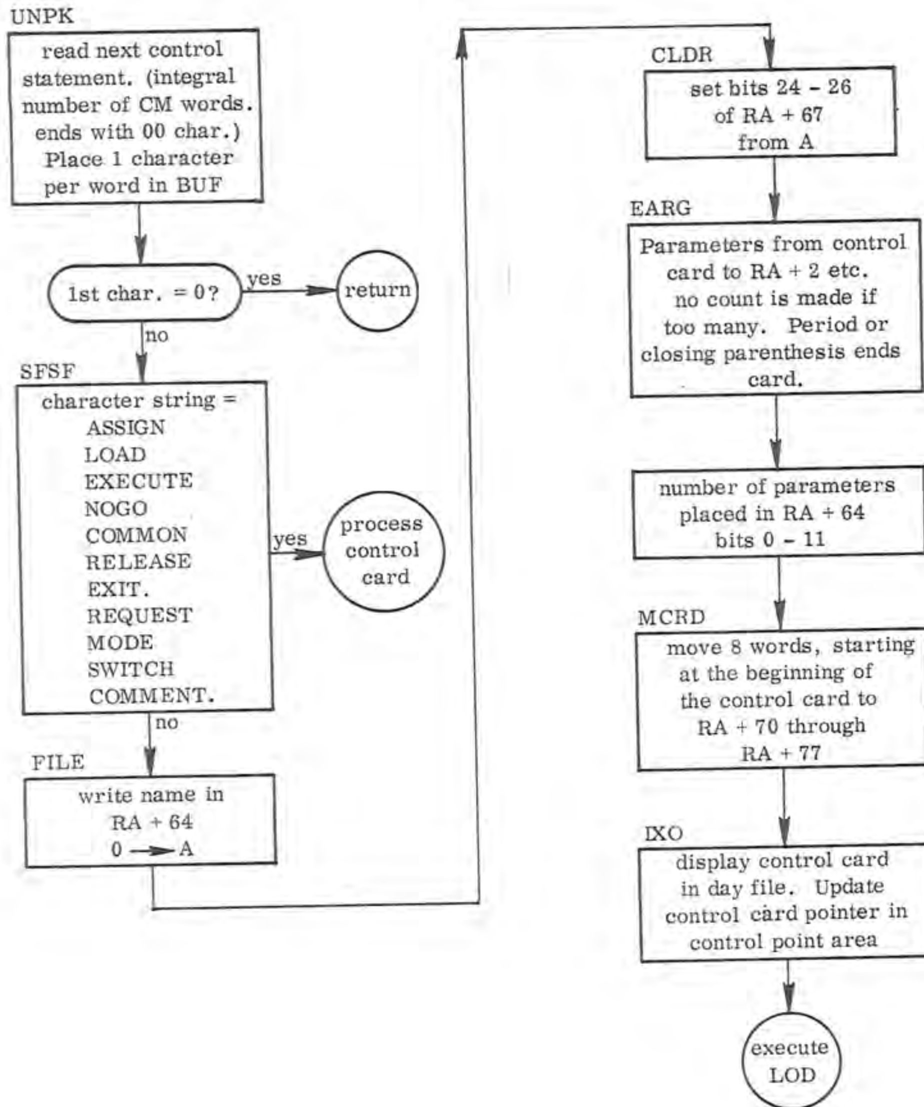
Location					
RA	0			α	Sense switches
+1			0		
+2 . . . +63			LOAD PARAMETERS		
+64	PROGRAM NAME				no. of parameters
+65		β			FWA for loading
+66		LWA for new tables	ζ	γ	FWA of program
+67		δ	ε		FWA of LOADER
+70 . . . +77			CONTROL CARD IMAGE		

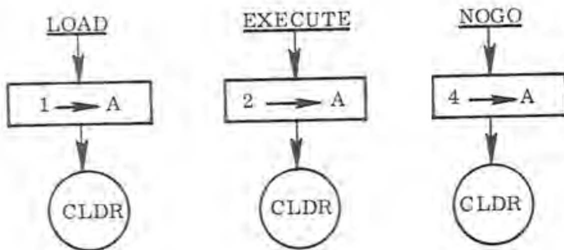
-ζ									current overlay level		Last overlay level	
α												pause bit
γ	Write overlay		ignore SEGMENT cards	OVERLAY flag	SEGMENT flag	SECTION flag	not part of γ					
ε					Partial map flag	end of load flag	exit REQUEST routine	MAP flag	RSS flag	NOGO card	EXECUTE card	LOAD card

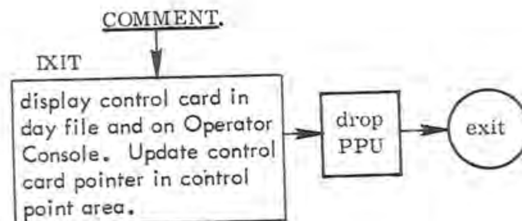
NOGO card, EXECUTE card, LOAD card: If none of these bits are set, then a program call card was encountered.

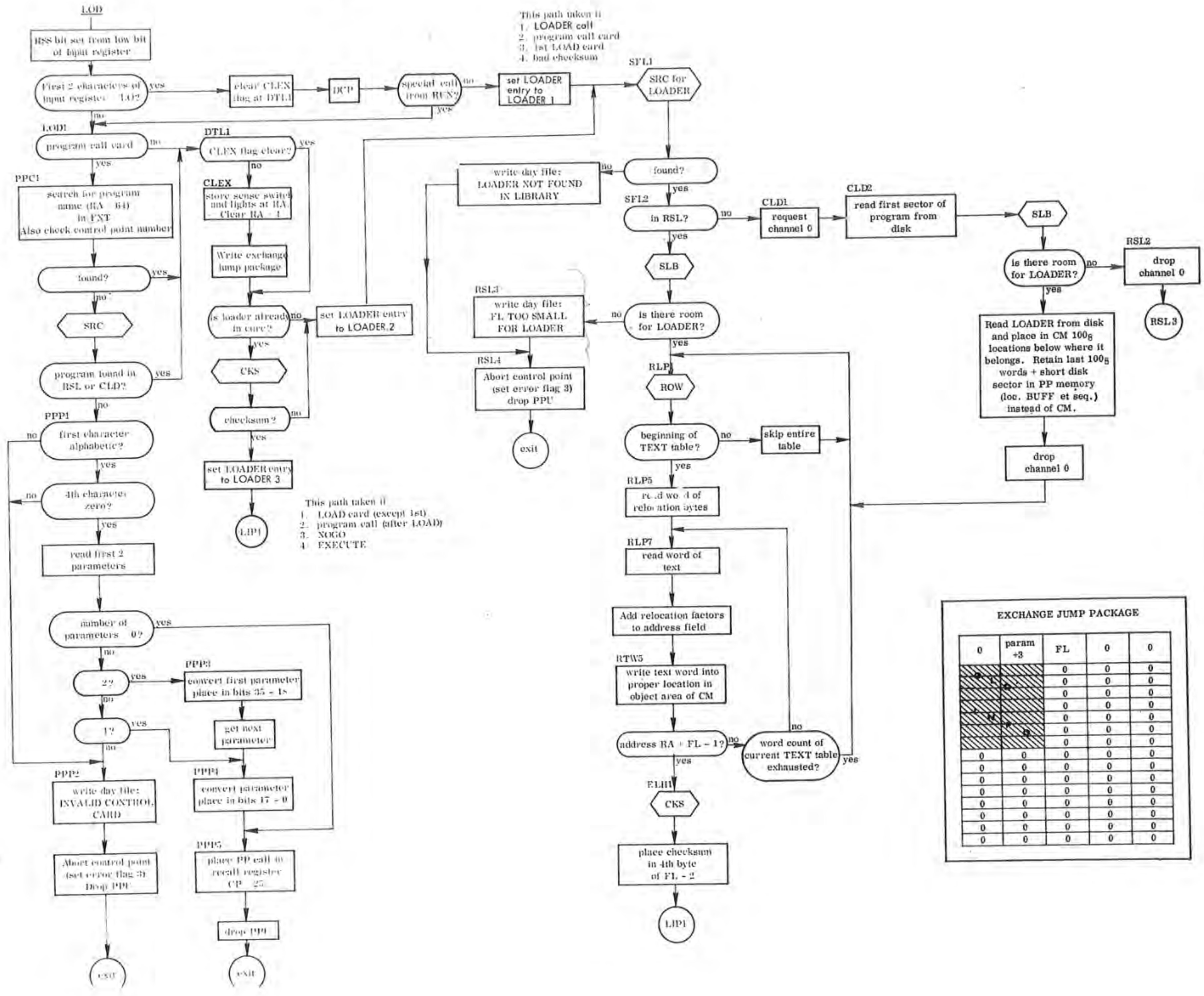
β: during Segment loading contains the FWA of the segment tables
during Overlay loading contains the FWA of OVERLOAD

δ: during Segment loading contains the FWA of the lowest load-link table
during Overlay loading contains the FWA of blank common.

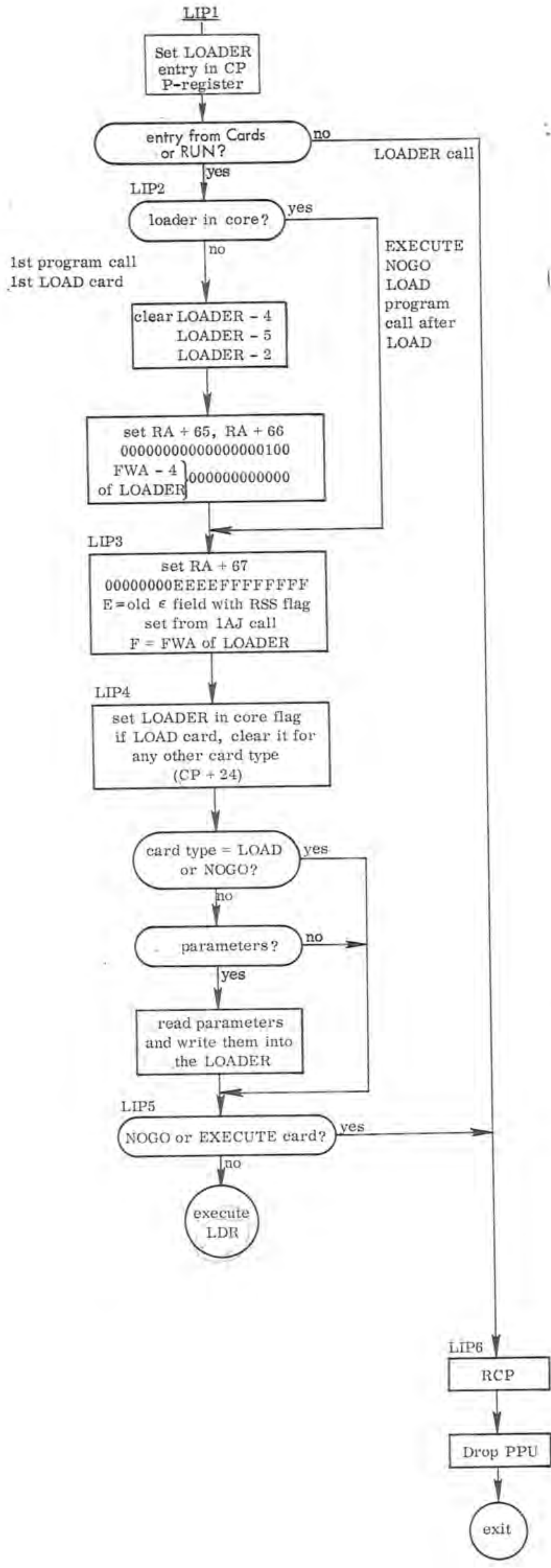

UNPK
read next control statement. (integral number of CM words. ends with 00 char.) Place 1 character per word in BUF
1st char. = 0?
yes
return
no
SFSF
character string =
ASSIGN
LOAD
EXECUTE
NOGO
COMMON
RELEASE
EXIT.
REQUEST
MODE
SWITCH
COMMENT.
yes
process control card
no
FILE
write name in RA + 64
0 → A
CLDR
set bits 24 - 26 of RA + 67 from A
EARG
Parameters from control card to RA + 2 etc. no count is made if too many. Period or closing parenthesis ends card.
number of parameters placed in RA + 64 bits 0 - 11
MCRD
move 8 words, starting at the beginning of the control card to RA + 70 through RA + 77
IXO
display control card in day file. Update control card pointer in control point area
execute LOD


LOAD
1 → A
CLDR
EXECUTE
2 → A
CLDR
NOGO
4 → A
CLDR


COMMENT.
IXIT
display control card in day file and on Operator Console. Update control card pointer in control point area.
drop PPU
exit



LOD
LOD
RSS bit set from low bit of input register
First 2 characters of input register = LO?
yes
no
clear CLEX flag at DTL1
DCP
special call from RUN?
no
yes
set LOADER entry to LOADER 1
This path taken if
1. LOADER call
2. program call card
3. 1st LOAD card
4. bad checksum
SFL1
SRC for LOADER
LOD1
program call card
no
yes
DTL1
CLEX flag clear?
yes
no
CLEX
store sense switch and lights at RA. Clear RA + 1
Write exchange jump package
is loader already in core?
no
yes
set LOADER entry to LOADER 2
CKS
checksum?
no
yes
set LOADER entry to LOADER 3
LIP1
This path taken if
1. LOAD card (except 1st)
2. program call (after LOAD)
3. NOGO
4. EXECUTE
PPC1
search for program name (RA + 64) in FXT
Also check control point number
found?
yes
no
SRC
program found in RSL or CLD?
yes
no
PPP1
first character alphabetic?
no
yes
4th character zero?
no
yes
read first 2 parameters
number of parameters = 0?
yes
no
2?
yes
no
PPP3
convert first parameter place in bits 35 - 18
get next parameter
1?
yes
no
PPP4
convert parameter place in bits 17 - 0
PPP5
place PP call in recall register CP + 25
drop PPU
exit
PPP2
write day file: INVALID CONTROL CARD
Abort control point (set error flag 3) Drop PPU
exit
found?
no
yes
write day file: LOADER NOT FOUND IN LIBRARY
SFL2
in RSL?
no
yes
CLD1
request channel 0
CLD2
read first sector of program from disk
SLB
is there room for LOADER?
no
yes
RSL2
drop channel 0
RSL3
Read LOADER from disk and place in CM 100$_8$ locations below where it belongs. Retain last 100$_8$ words + short disk sector in PP memory (loc. BUFF et seq.) instead of CM.
drop channel 0
SLB
is there room for LOADER?
no
yes
RSL3
write day file: FL TOO SMALL FOR LOADER
RSL4
Abort control point (set error flag 3) drop PPU
exit
RLP1
ROW
beginning of TEXT table?
no
yes
skip entire table
RLP5
read word of relocation bytes
RLP7
read word of text
Add relocation factors to address field
RTW5
write text word into proper location in object area of CM
address RA + FL - 1?
no
yes
word count of current TEXT table exhausted?
no
yes
ELH1
CKS
place checksum in 4th byte of FL - 2
LIP1
EXCHANGE JUMP PACKAGE
0
param +3
FL
0
0

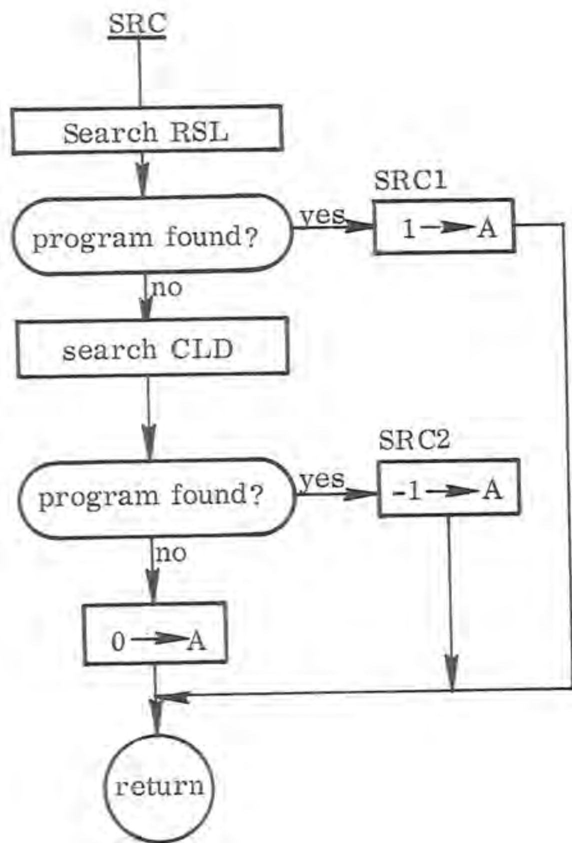


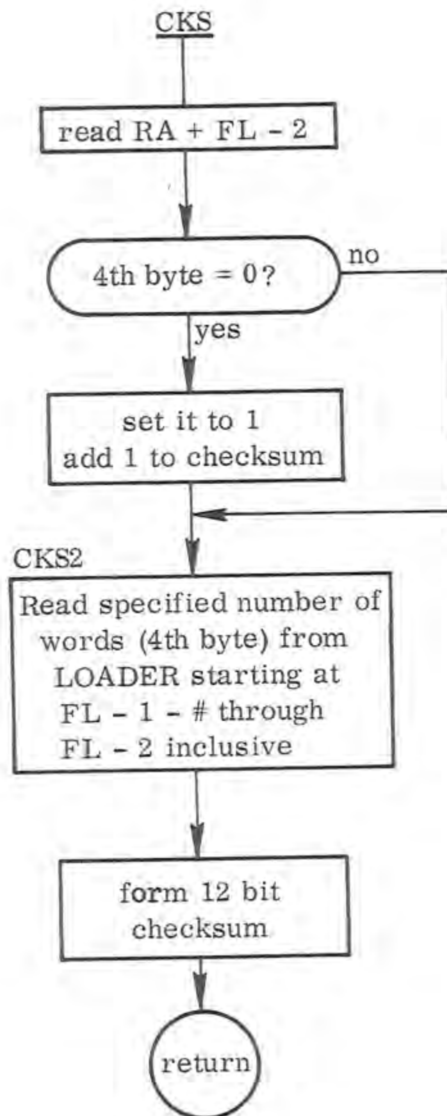
LIP1
Set LOADER entry in CP P-register
entry from Cards or RUN?
no
LOADER call
yes
LIP2
loader in core?
yes
no
1st program call
1st LOAD card
EXECUTE
NOGO
LOAD
program
call after
LOAD
clear LOADER - 4
LOADER - 5
LOADER - 2
set RA + 65, RA + 66
00000000000000000100
FWA - 4 of LOADER
000000000000
LIP3
set RA + 67
00000000EEEEFFFFFFFF
E = old ϵ field with RSS flag set from 1AJ call
F = FWA of LOADER
LIP4
set LOADER in core flag if LOAD card, clear it for any other card type
(CP + 24)
card type = LOAD or NOGO?
yes
no
parameters?
no
yes
read parameters and write them into the LOADER
LIP5
NOGO or EXECUTE card?
yes
no
execute LDR
LIP6
RCP
Drop PPU
exit

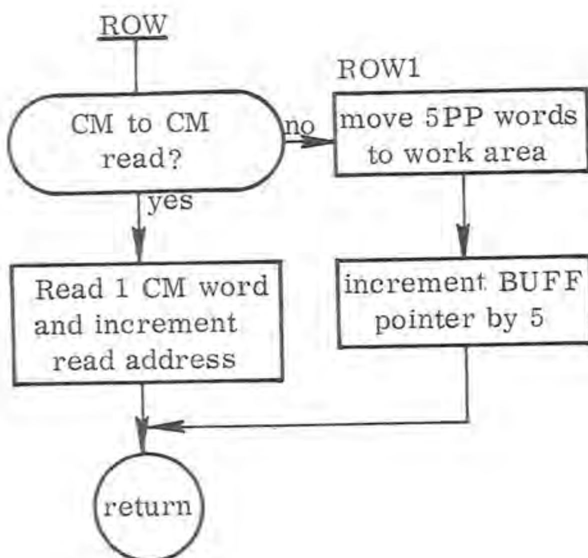
Set LOADER entry in CP P-register

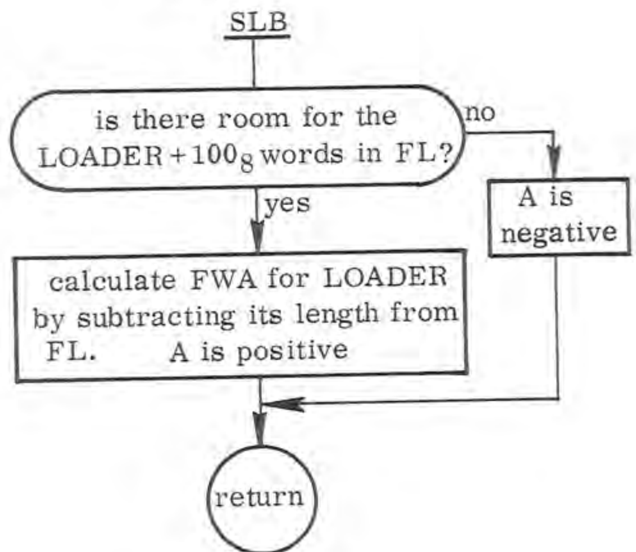
entry from Cards or RUN?

Yes


SRC
Search RSL
program found?
yes
SRC1
1→A
no
search CLD
program found?
yes
SRC2
-1→A
no
0→A
return


CKS
read RA + FL - 2
4th byte = 0?
no
yes
set it to 1
add 1 to checksum
CKS2
Read specified number of words (4th byte) from LOADER starting at FL - 1 - # through FL - 2 inclusive
form 12 bit checksum
return


ROW
CM to CM read?
no
ROW1
move 5PP words to work area
yes
Read 1 CM word and increment read address
increment BUFF pointer by 5
return


SLB
is there room for the LOADER + 100_8 words in FL?
no
A is negative
yes
calculate FWA for LOADER by subtracting its length from FL. A is positive
return

INIT

INIT
entry from LOADER? — yes → USERCAL: DCP → read the 2 word call → FWA = 0? — no → set FWA from call; yes → NOFWA: LWA = 0? — no → set LWA from call; yes → NOLWADR: save U, S & V bits & γ bits (RA+66) in STATFLG

NOLWADR → FN field = 0? — no → save file name, set FN bit in STATFLG; yes →

SLFLG: save SL field → = 0? — no → set SL bit in STATFLG; yes →

V bit set? — yes → OVLCAL; no →

UETST: U bit set? — YES → UEXCAL; no →

FNTST: FN bit set? — no → OVGEN: Read Source File name from LOADER → PASRAB; yes →

SL bit set? — no → FNCAL: set rewind bit for READ subroutine → PASRAB; yes →

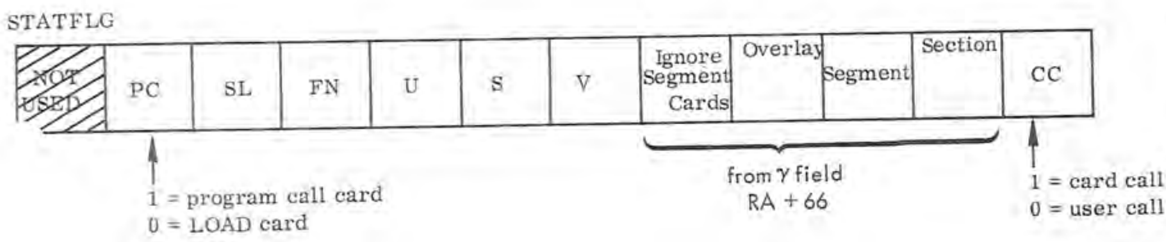
SLNQZRO: SL list entry = 0? — no → FNSLCAL; yes →

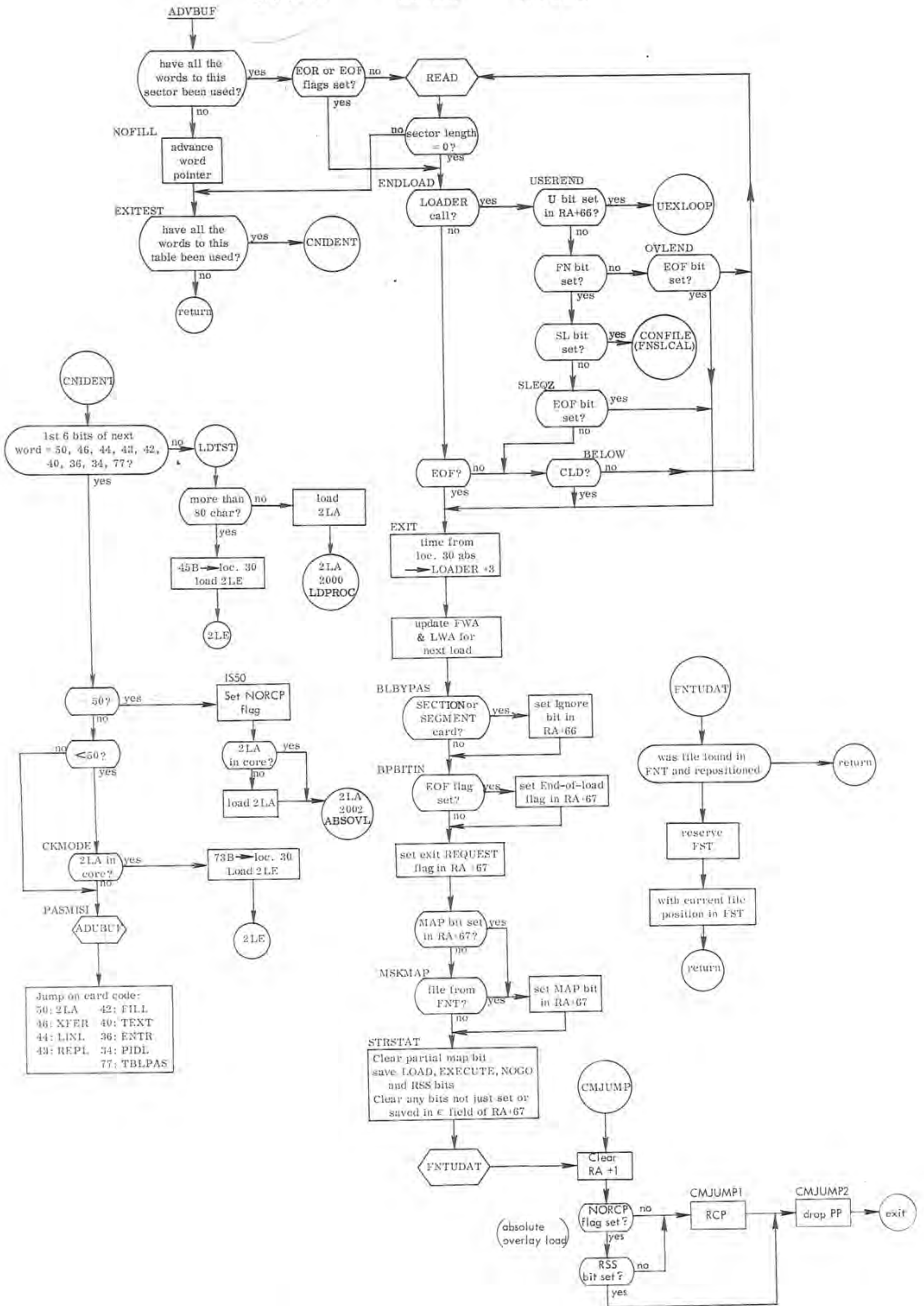
SLZERO: set exit REQUEST routine flag in RA + 67 → CMJUMP

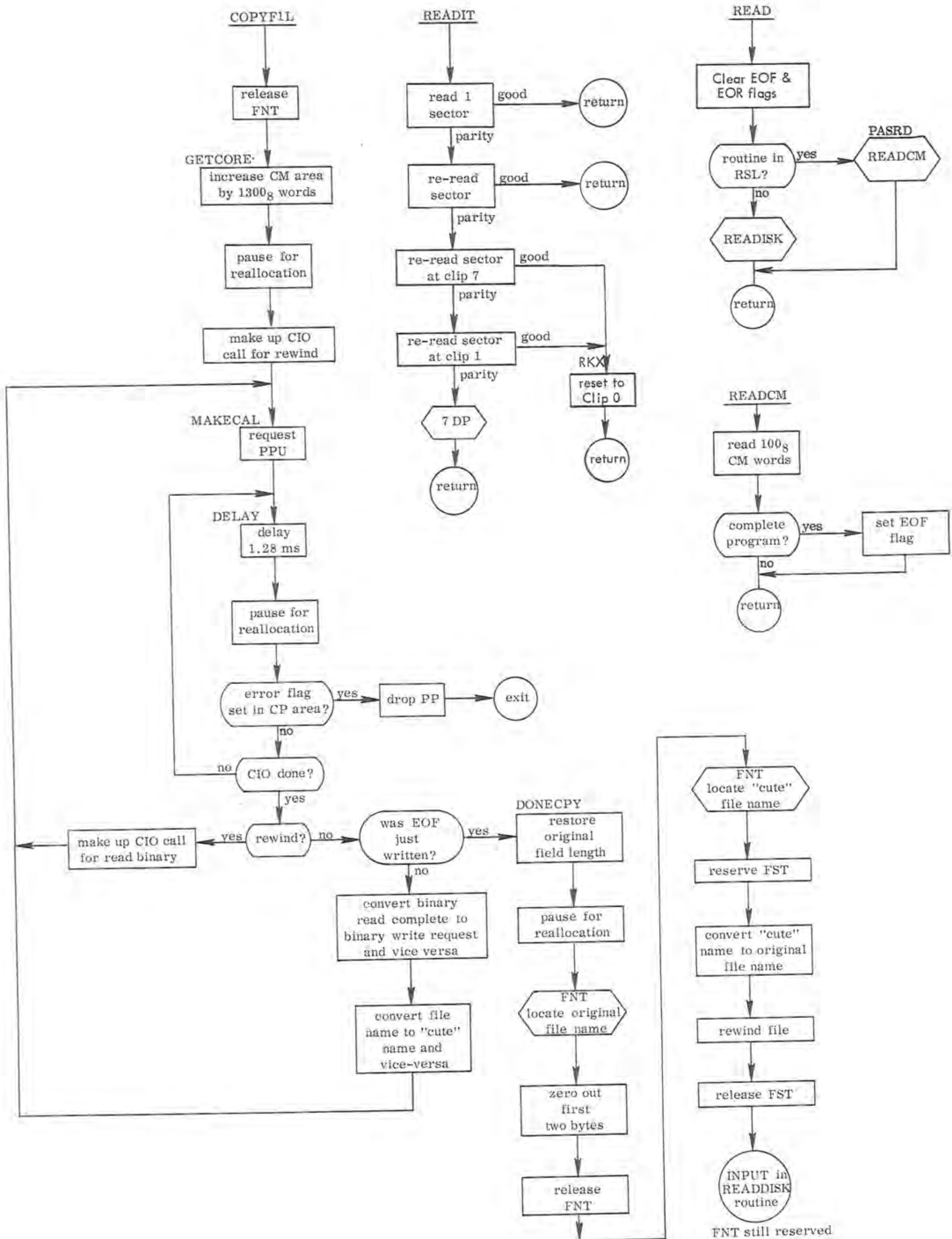
entry from LOADER? — no → set low bit in STATFLG → LOAD card? — yes → LOAD: read file name from RA +2 → PASRAB; no → set prog. call bit in STATFLG → read file name from RA +64 →

PASRAB: FNT → found? — yes → FOUNDX; no → RSLCLD → found? — no → 41B → loc. 30 Load 2 LE → 2LE; yes →

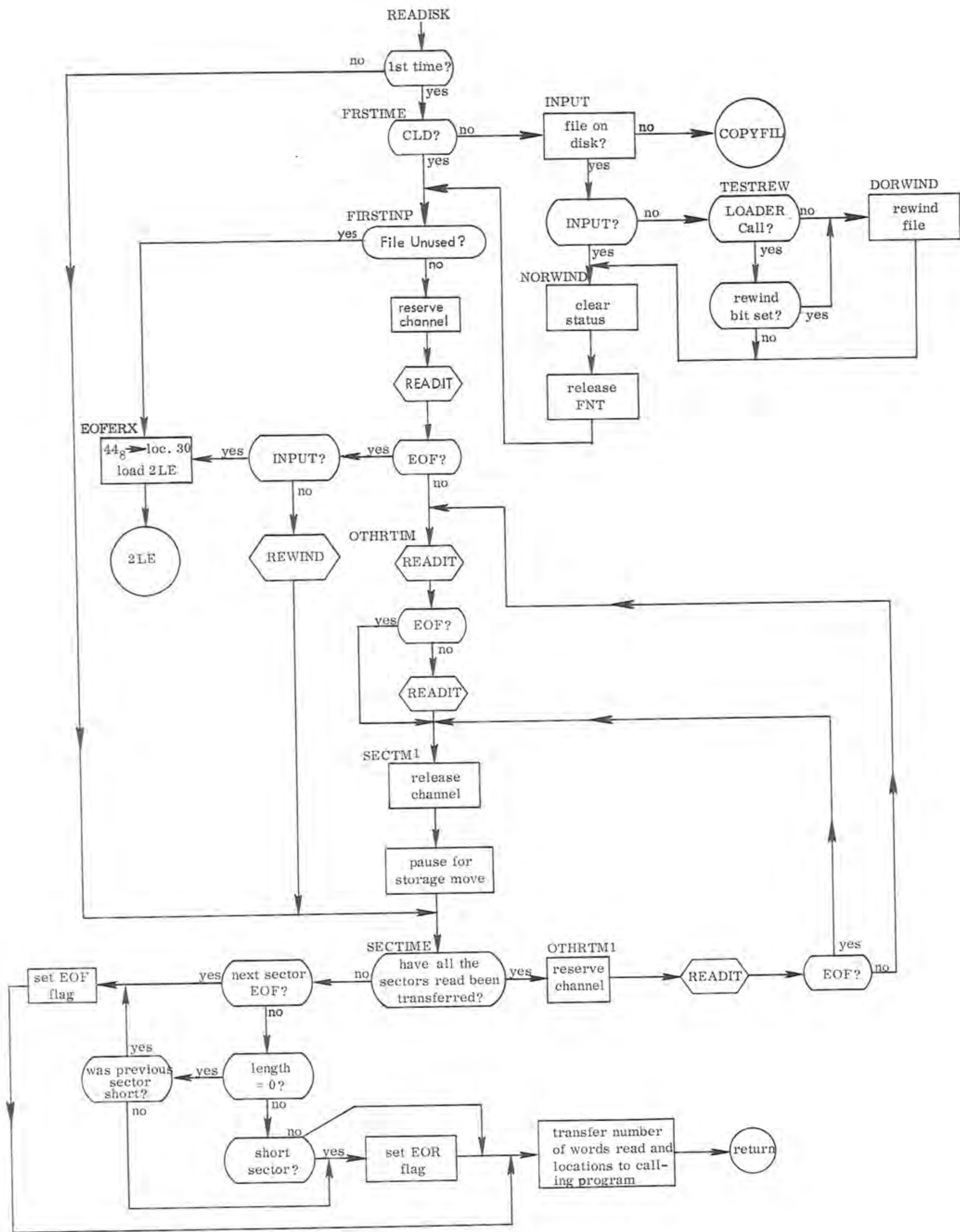
FOUNDX: READ → CNIDENT



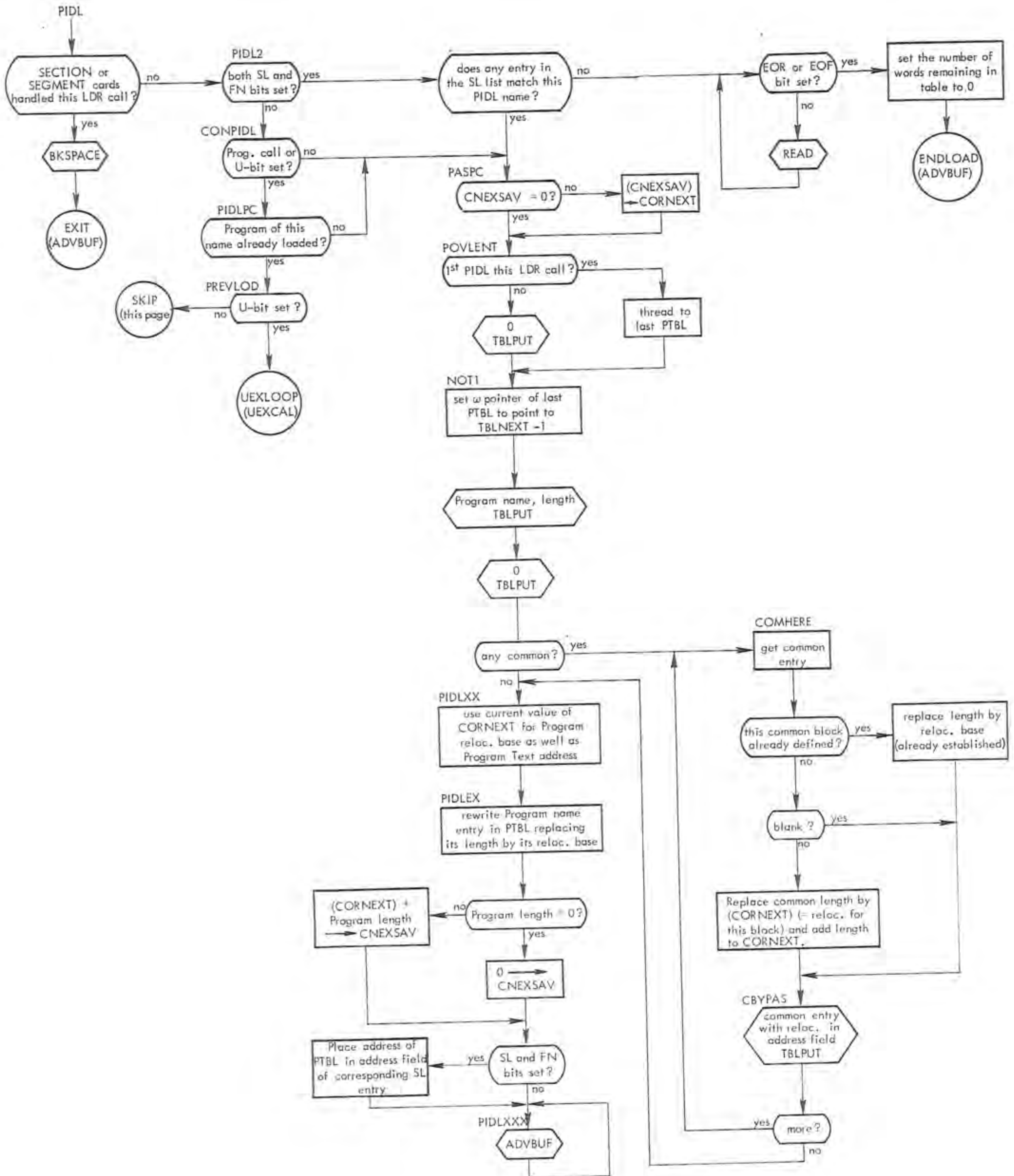

ADVBUF
have all the words to this sector been used?
yes
no
EOR or EOF flags set?
READ
sector length = 0?
NOFILL
advance word pointer
ENDLOAD
LOADER call?
USEREND
U bit set in RA+66?
UEXLOOP
EXITEST
have all the words to this table been used?
CNIDENT
return
OVLEND
FN bit set?
EOF bit set?
SL bit set?
CONFILE (FNSLCAL)
SLEQZ
EOF bit set?
CNIDENT
1st 6 bits of next word = 50, 46, 44, 43, 42, 40, 36, 34, 77?
LDTST
BELOW
EOF?
CLD?
more than 80 char?
load 2LA
EXIT
time from loc. 30 abs → LOADER +3
45B → loc. 30 load 2LE
2LA 2000 LDPROC
2LE
update FWA & LWA for next load
IS50
Set NORCP flag
BLBYPAS
SECTION or SEGMENT card?
set Ignore bit in RA+66
FNTUDAT
50?
2LA in core?
<50?
BPBITIN
EOF flag set?
set End-of-load flag in RA+67
was file found in FNT and repositioned
return
load 2LA
2LA 2002 ABSOVL
reserve FST
CKMODE
2LA in core?
73B → loc. 30 Load 2LE
set exit REQUEST flag in RA+67
with current file position in FST
PASMISI
ADVBUF
2LE
MAP bit set in RA+67?
return
MSKMAP
file from FNT?
set MAP bit in RA+67
Jump on card code:
50: 2LA 42: FILL
46: XFER 40: TEXT
44: LINK 36: ENTR
43: REPL 34: PIDL
77: TBLPAS
STRSTAT
Clear partial map bit
save LOAD, EXECUTE, NOGO and RSS bits
Clear any bits not just set or saved in ε field of RA+67
CMJUMP
FNTUDAT
Clear RA +1
CMJUMP1
CMJUMP2
NORCP flag set?
RCP
drop PP
exit
(absolute overlay load)
RSS bit set?

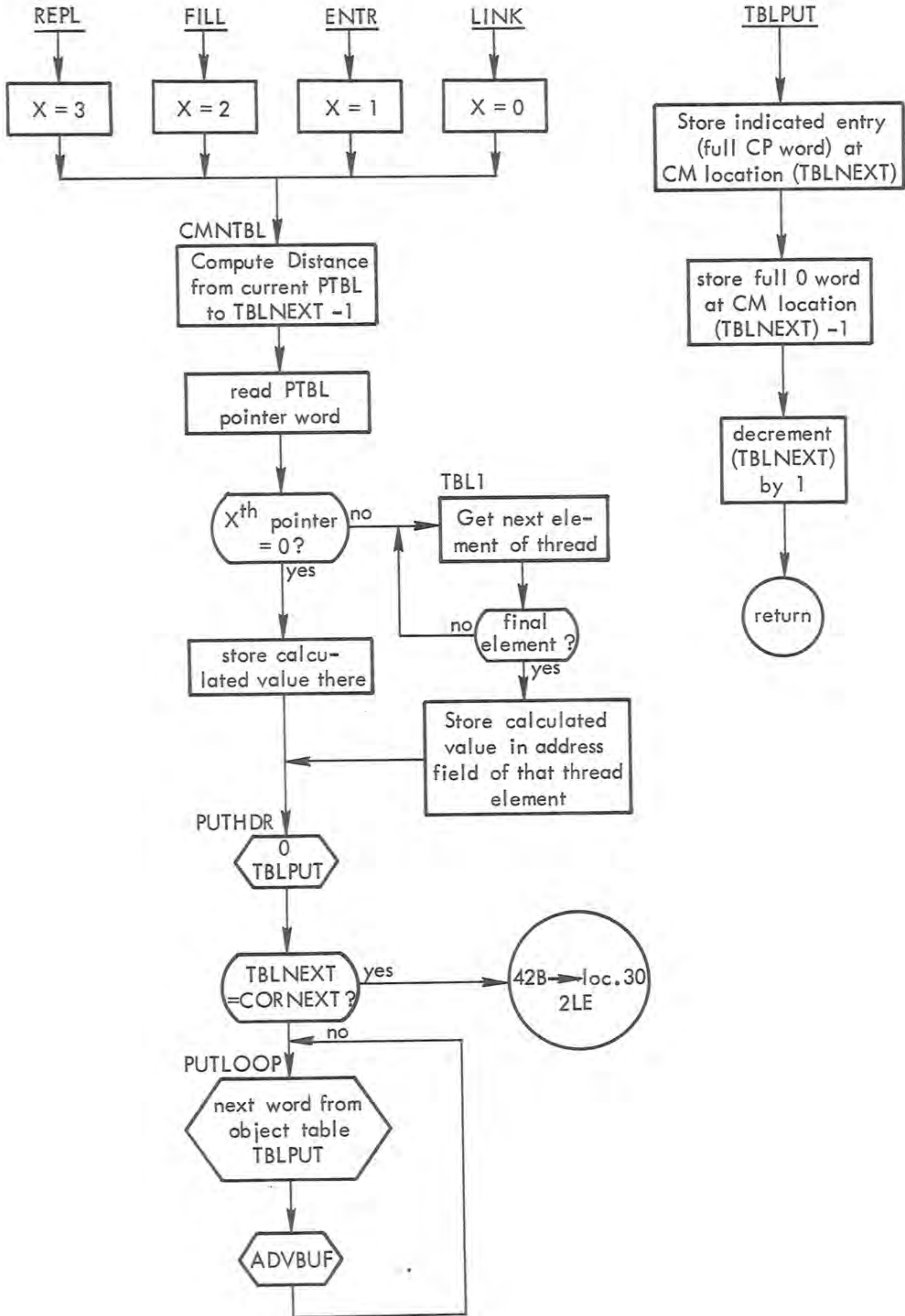


COPYF1L
release FNT
GETCORE
increase CM area by 1300$_8$ words
pause for reallocation
make up CIO call for rewind
MAKECAL
request PPU
DELAY
delay 1.28 ms
pause for reallocation
error flag set in CP area?
yes
drop PP
exit
no
CIO done?
no
yes
rewind?
yes
make up CIO call for read binary
no
was EOF just written?
no
convert binary read complete to binary write request and vice versa
convert file name to "cute" name and vice-versa
yes
DONECPY
restore original field length
pause for reallocation
FNT locate original file name
zero out first two bytes
release FNT
FNT locate "cute" file name
reserve FST
convert "cute" name to original file name
rewind file
release FST
INPUT in READDISK routine
FNT still reserved
READIT
read 1 sector
good
return
parity
re-read sector
good
return
parity
re-read sector at clip 7
good
parity
re-read sector at clip 1
good
parity
7 DP
return
RKX
reset to Clip 0
return
READ
Clear EOF & EOR flags
routine in RSL?
yes
PASRD
READCM
no
READISK
return
READCM
read 100$_8$ CM words
complete program?
yes
set EOF flag
no
return


READISK
1st time?
no
yes
FRSTIME
CLD?
no
yes
INPUT
file on disk?
no
COPYFIL
yes
INPUT?
no
TESTREW
LOADER Call?
no
DORWIND
rewind file
yes
rewind bit set?
yes
no
yes
NORWIND
clear status
release FNT
FIRSTINP
File Unused?
yes
no
reserve channel
READIT
EOFERX
44_8 → loc. 30
load 2LE
yes
INPUT?
yes
EOF?
no
no
2LE
REWIND
OTHRTIM
READIT
EOF?
yes
no
READIT
SECTM1
release channel
pause for storage move
SECTIME
have all the sectors read been transferred?
yes
OTHRTM1
reserve channel
READIT
EOF?
yes
no
no
next sector EOF?
yes
set EOF flag
no
length = 0?
yes
was previous sector short?
yes
no
no
short sector?
no
yes
set EOR flag
transfer number of words read and locations to calling program
return

PIDL





REPL
FILL
ENTR
LINK
X = 3
X = 2
X = 1
X = 0
CMNTBL
Compute Distance from current PTBL to TBLNEXT -1
read PTBL pointer word
X^th pointer = 0?
no
yes
TBL1
Get next element of thread
final element ?
no
yes
store calculated value there
Store calculated value in address field of that thread element
PUTHDR
0
TBLPUT
TBLNEXT =CORNEXT ?
yes
no
42B→loc.30
2LE
PUTLOOP
next word from object table
TBLPUT
ADVBUF
TBLPUT
Store indicated entry (full CP word) at CM location (TBLNEXT)
store full 0 word at CM location (TBLNEXT) -1
decrement (TBLNEXT) by 1
return

TBLPUT

Store indicated entry (full CP word) at CM location (TBLNEXT)

store full word at CM location (TBLNEXT) - 1

decrement (TBLNEXT) by

return

LINK

ENTR

FILL

REPL

X = 0

X = 1

X = 2

X = 3

CMNTBL

Compute distance from current PTBL to TBLNEXT - 1

add PTBL pointer word

X^th pointer = 0 ?

no

yes

TBL1

get next element of thread

final element ?

no

yes

Store calculated value in address field of final thread element

store calculated value there

TBLPUT

TBLNEXT < ORNEXT ?

yes

928 — loc. of 215

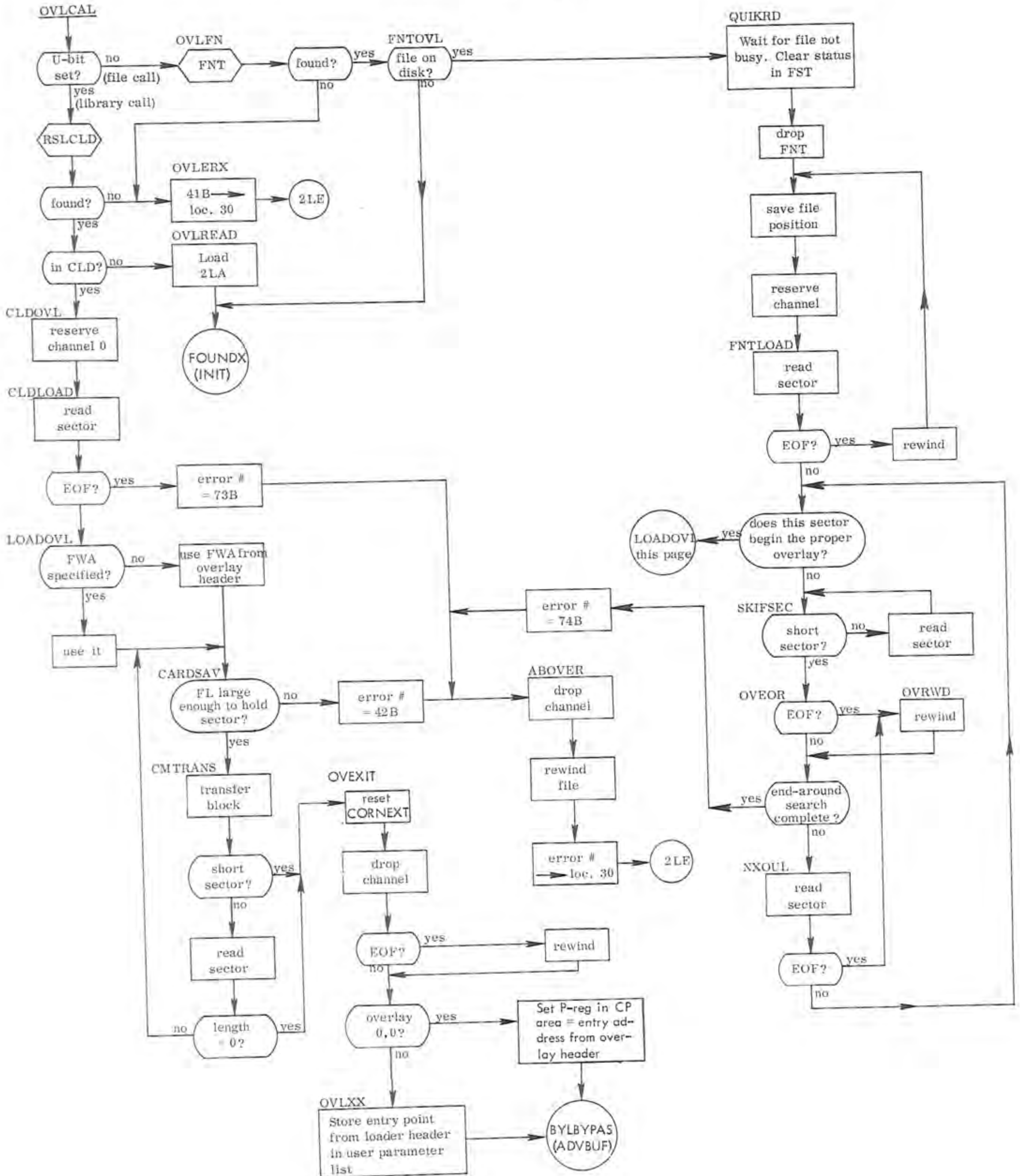
no

PUTLOOP

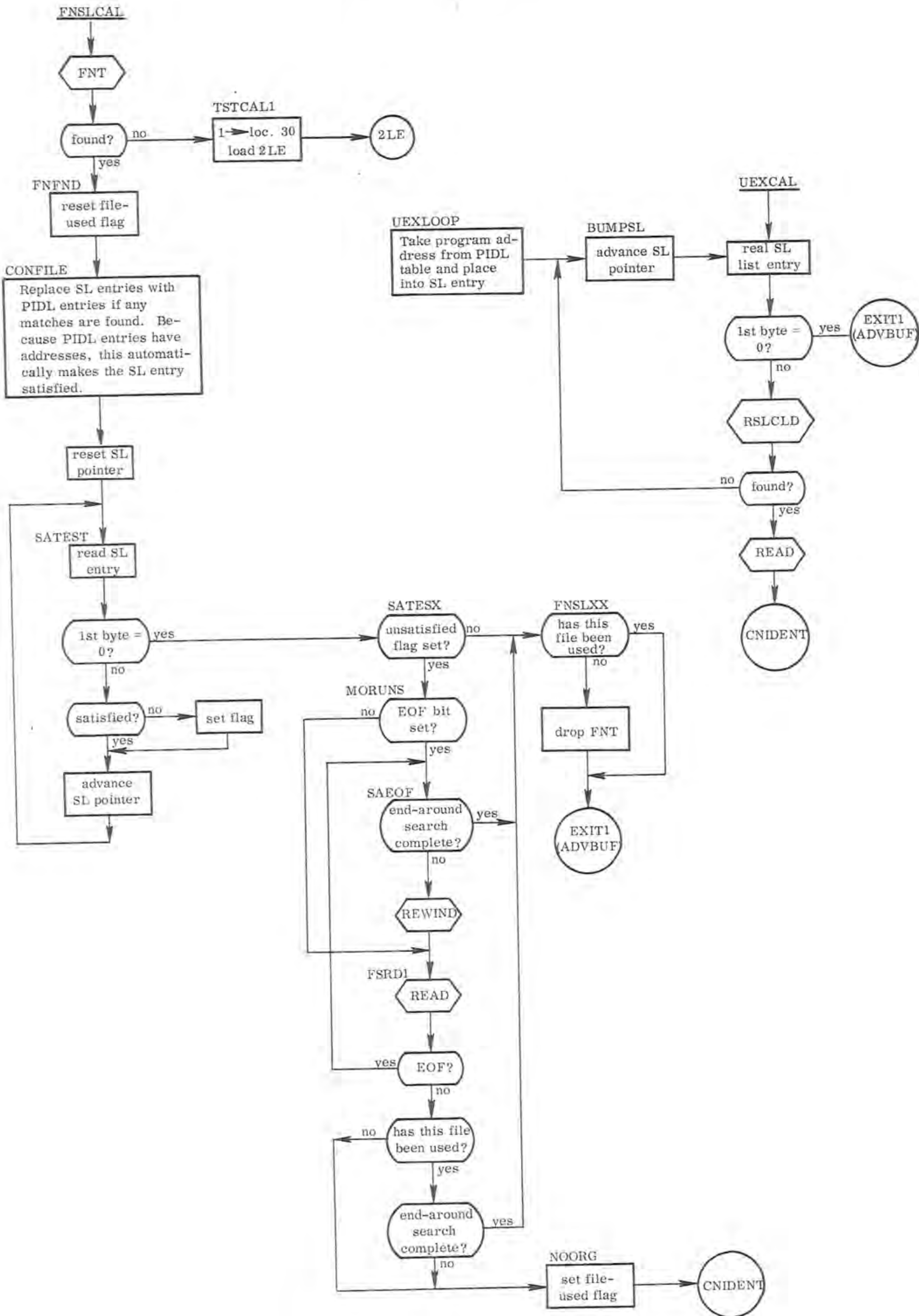
next word from object table TBLPUT

ADVBUF

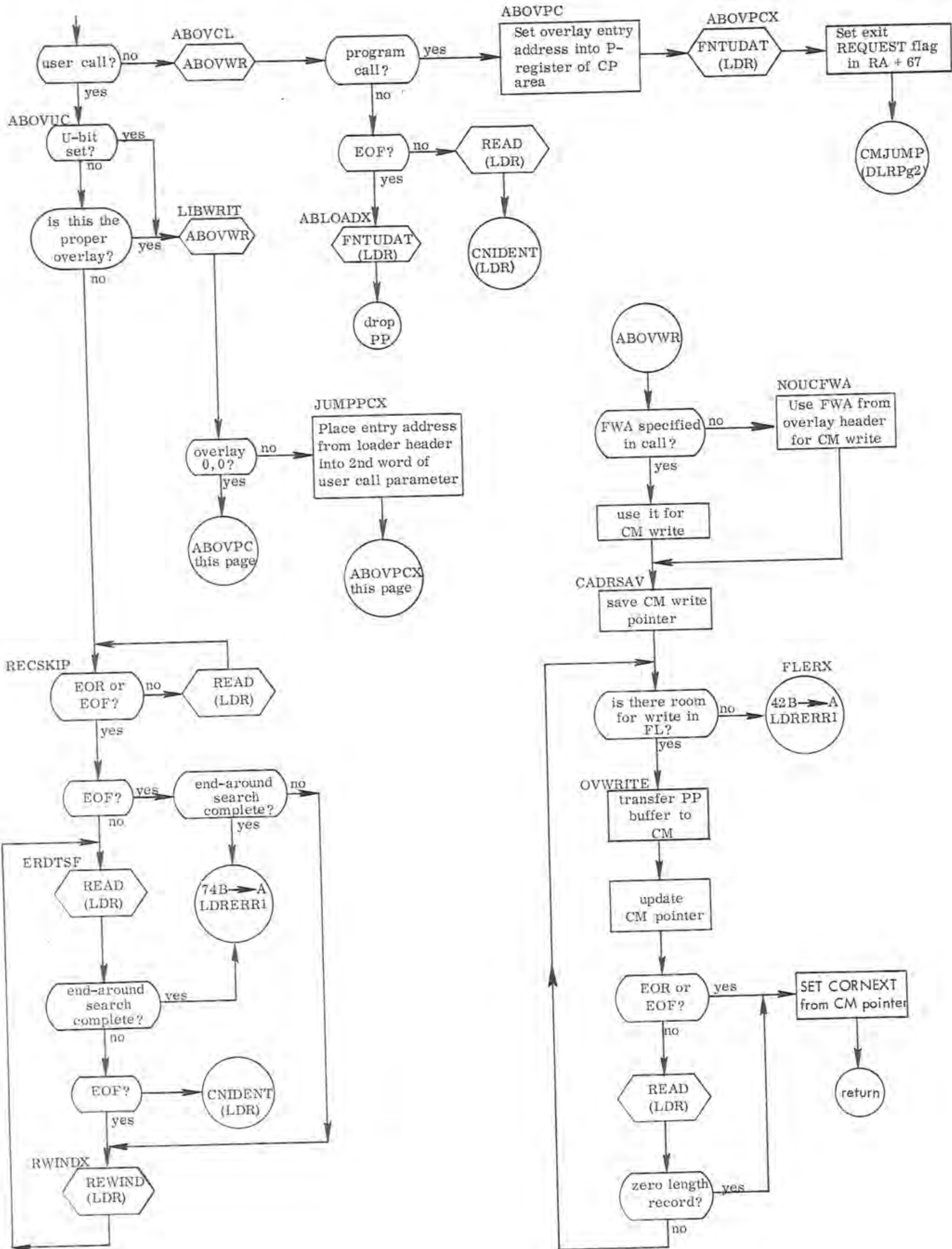
OVLCAL



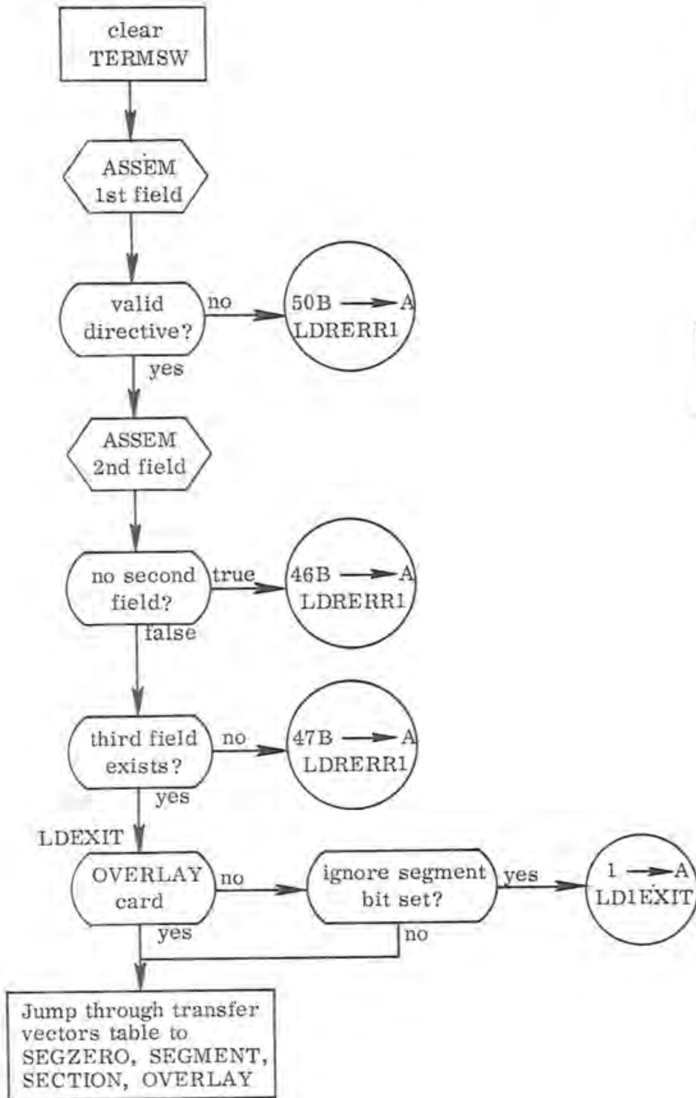
FNSLCAL and UEXCAL

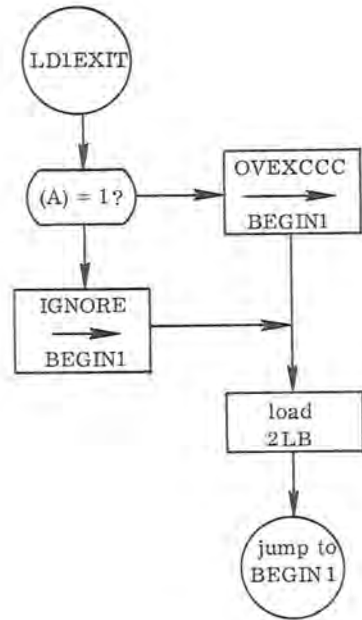


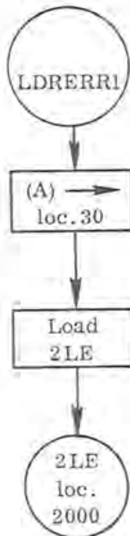
ABSOVL



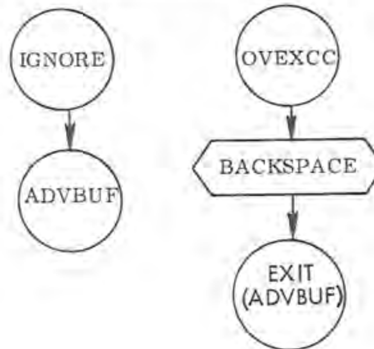
LDPROC

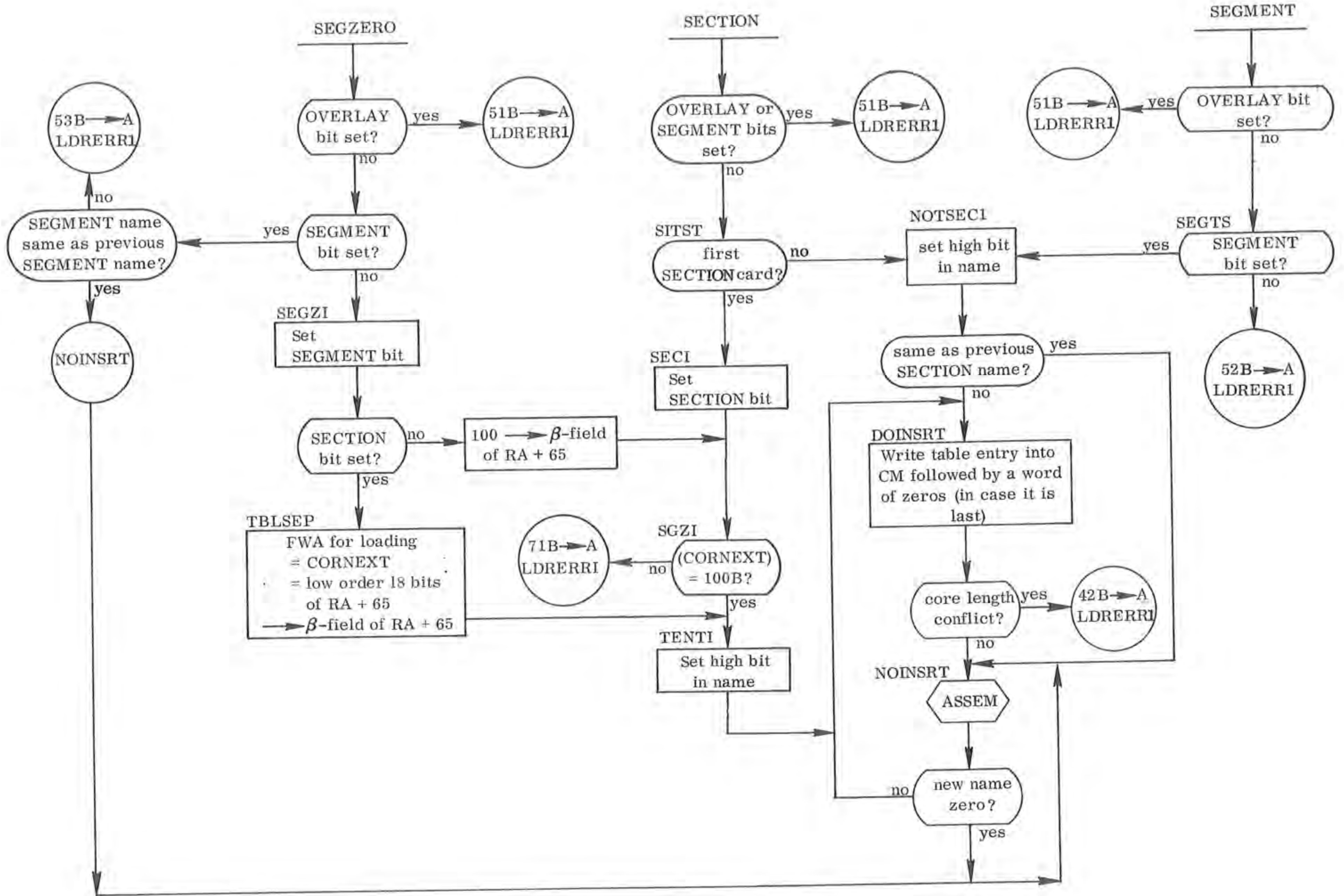






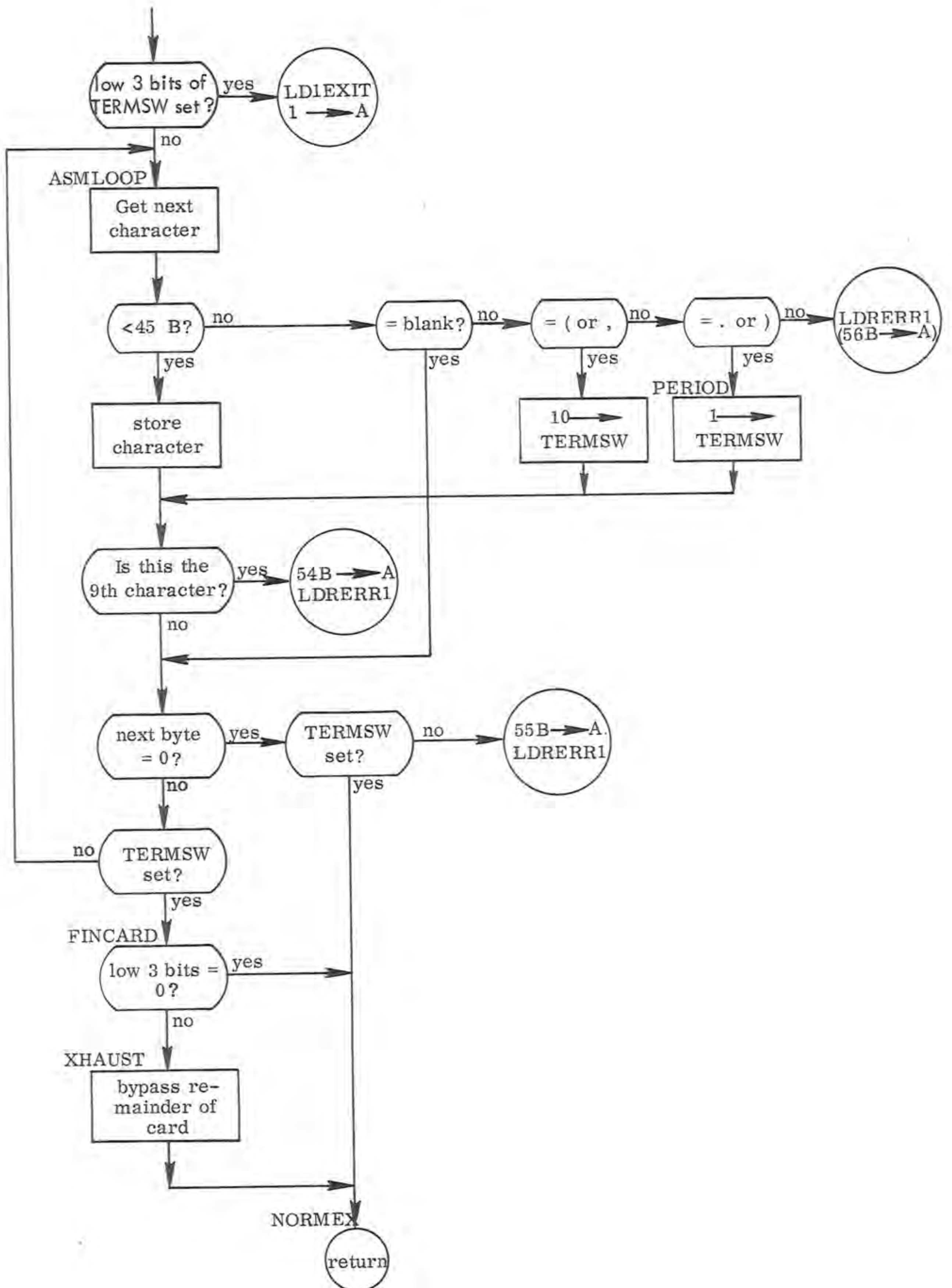
Routines in LDR



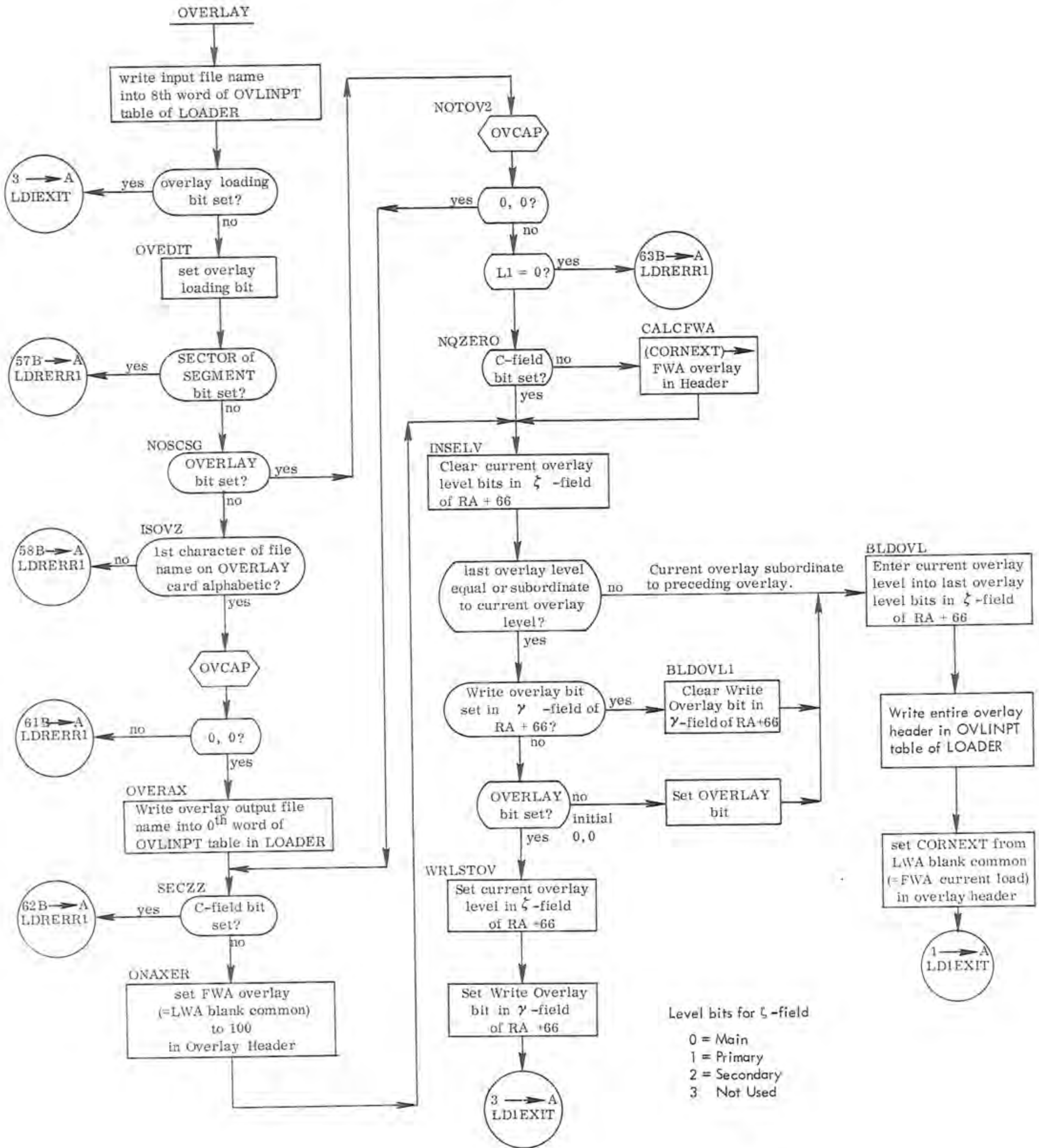


SEGZERO
OVERLAY bit set?
yes
51B → A LDRERR1
no
SEGMENT bit set?
yes
SEGMENT name same as previous SEGMENT name?
no
53B → A LDRERR1
yes
NOINSRT
no
SEGZI
Set SEGMENT bit
SECTION bit set?
no
100 → β-field of RA + 65
yes
TBLSEP
FWA for loading = CORNEXT = low order 18 bits of RA + 65 → β-field of RA + 65
SECTION
OVERLAY or SEGMENT bits set?
yes
51B → A LDRERR1
no
SITST
first SECTION card?
no
yes
SECI
Set SECTION bit
SGZI
(CORNEXT) = 100B?
no
71B → A LDRERRI
yes
TENTI
Set high bit in name
SEGMENT
OVERLAY bit set?
yes
51B → A LDRERR1
no
SEGTS
SEGMENT bit set?
no
52B → A LDRERR1
yes
NOTSEC1
set high bit in name
same as previous SECTION name?
yes
no
DOINSRT
Write table entry into CM followed by a word of zeros (in case it is last)
core length conflict?
yes
42B → A LDRERR1
no
NOINSRT
ASSEM
new name zero?
no
yes

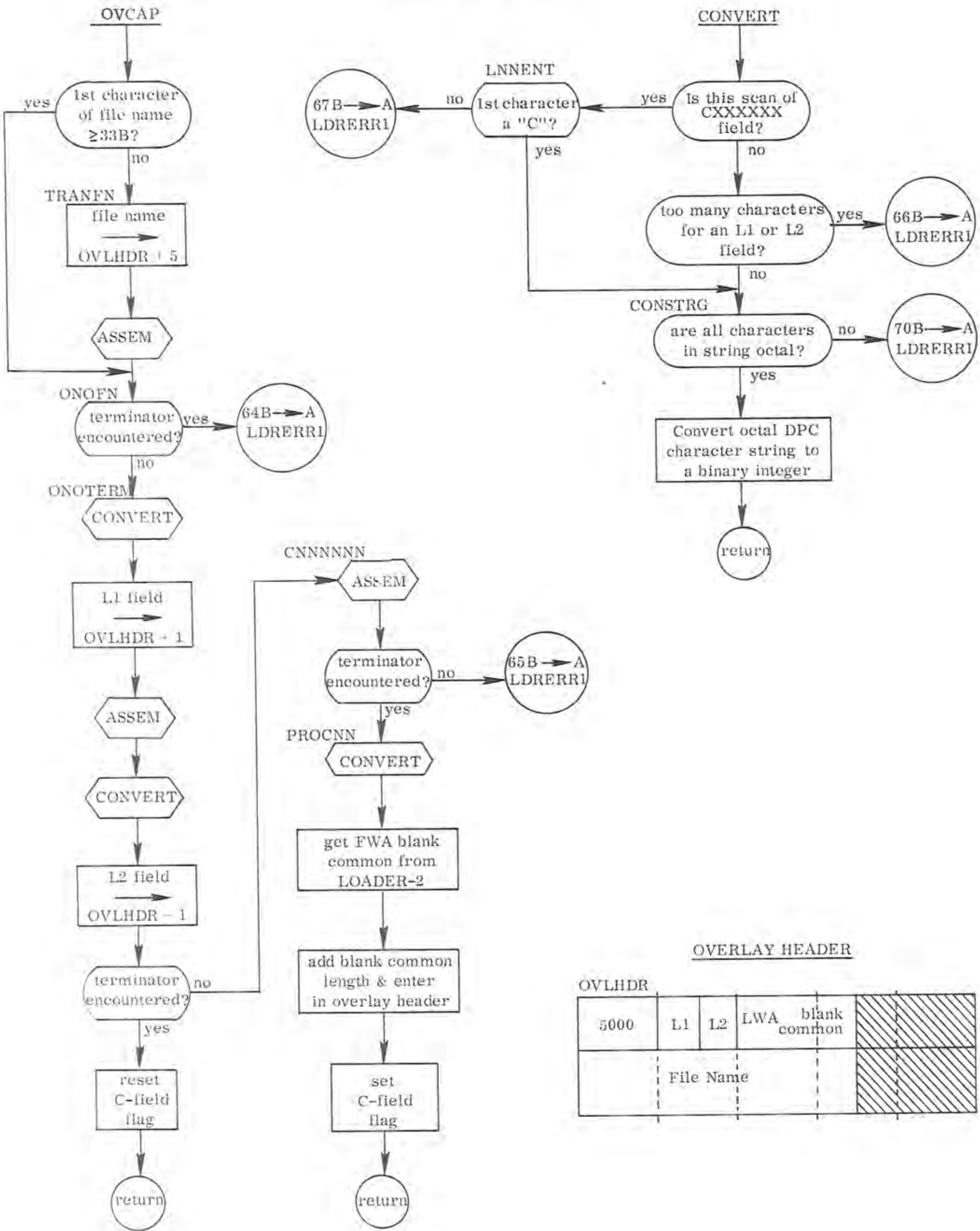
ASSEM



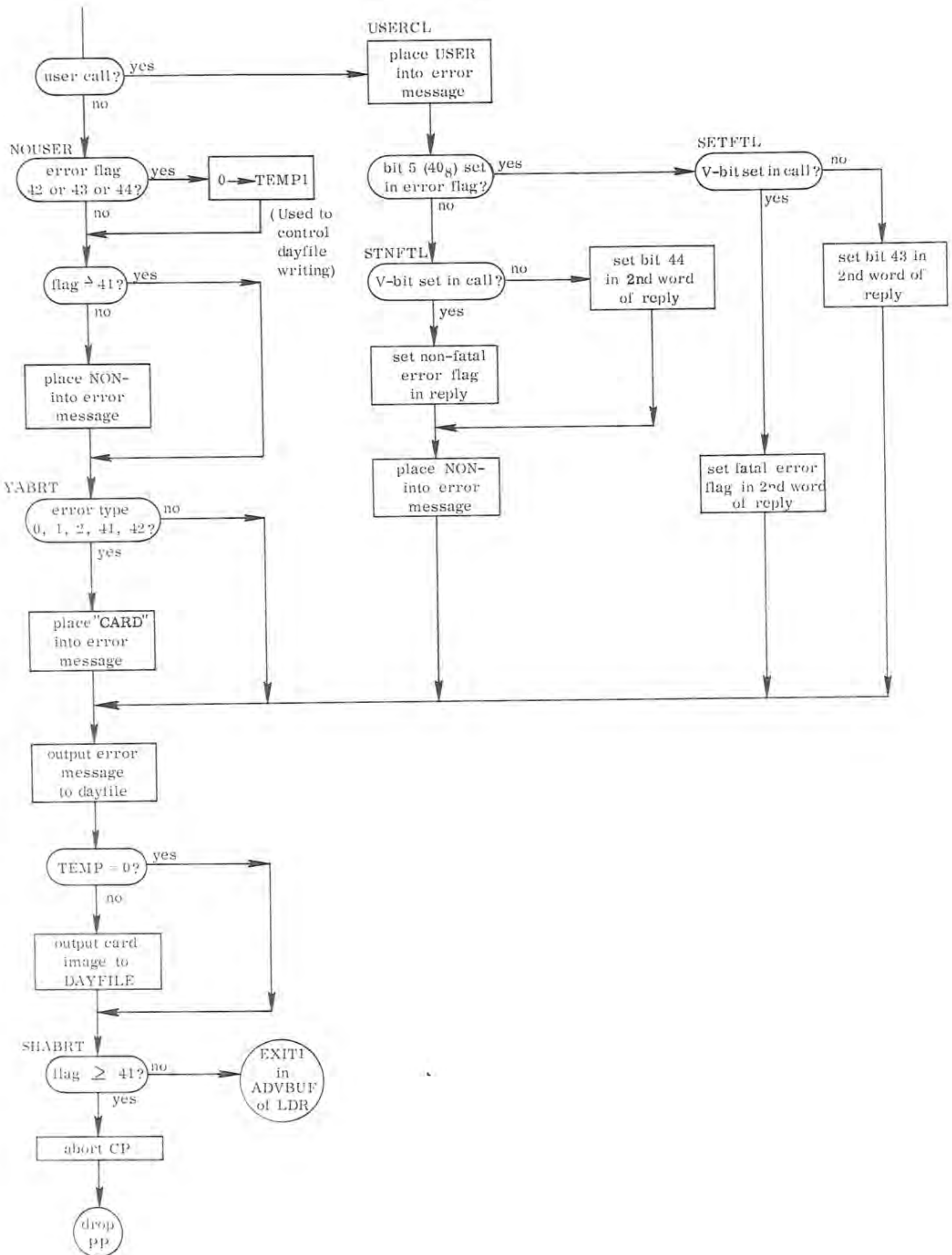
OVERLAY



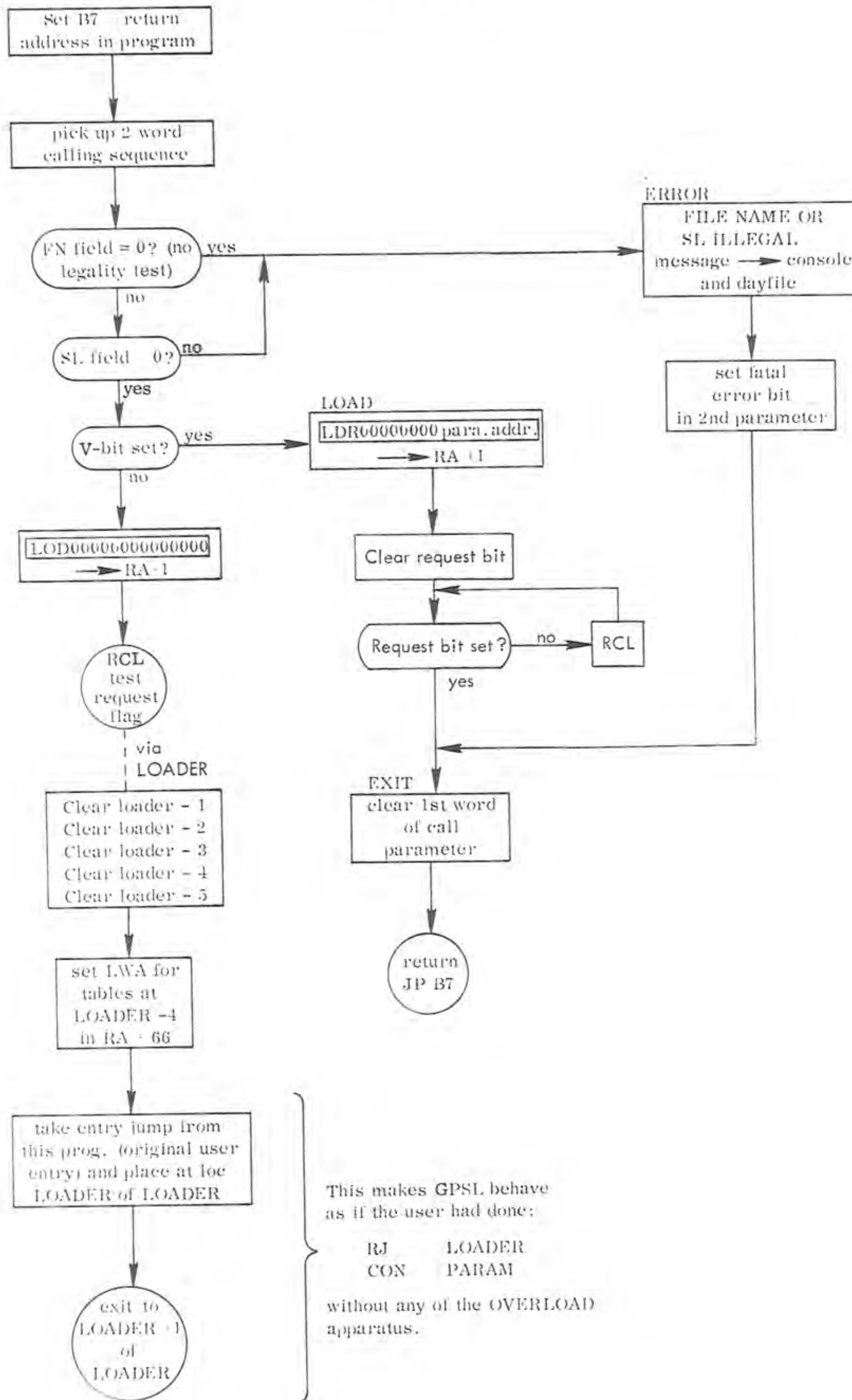
OVCAP & CONVERT



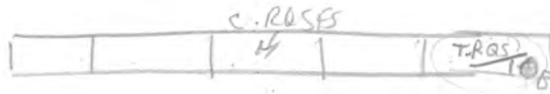
2 LE



OVERLOD



DST P.RQS (11)




```
LDN   P.RQS
CRD   D.TO
LDD   D.TO + C.RQSFS    (2)
SHN   3
CRD   —
```

max # of Requests	Not used	FWA requests	# Devices

cycle counter 4096₈

DSuit

REQUEST

2RC

end of job flag set?
yes
clear end of job flag
set EOF bit in FST entry
return
no

CRD1
modify overlay for equipment parameters

CRD2
determine buffer size for input

CRD4
room for 17 words?
no
return
yes

CRD5
RES

CRD6
clear console message

ROC
STS

read 80 card columns to BUF

clear EF

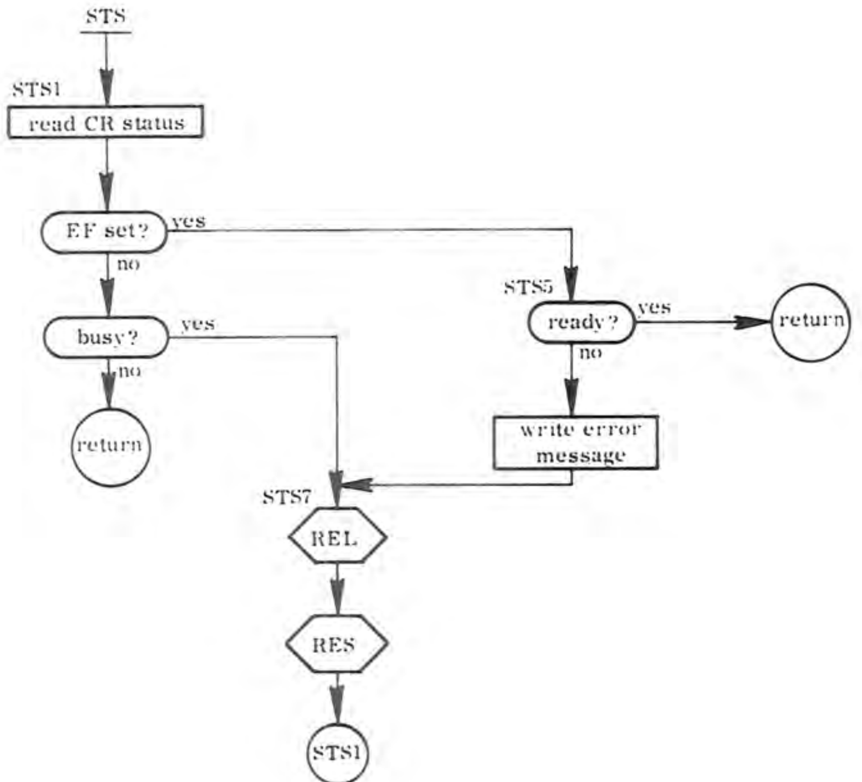
move status word to LS

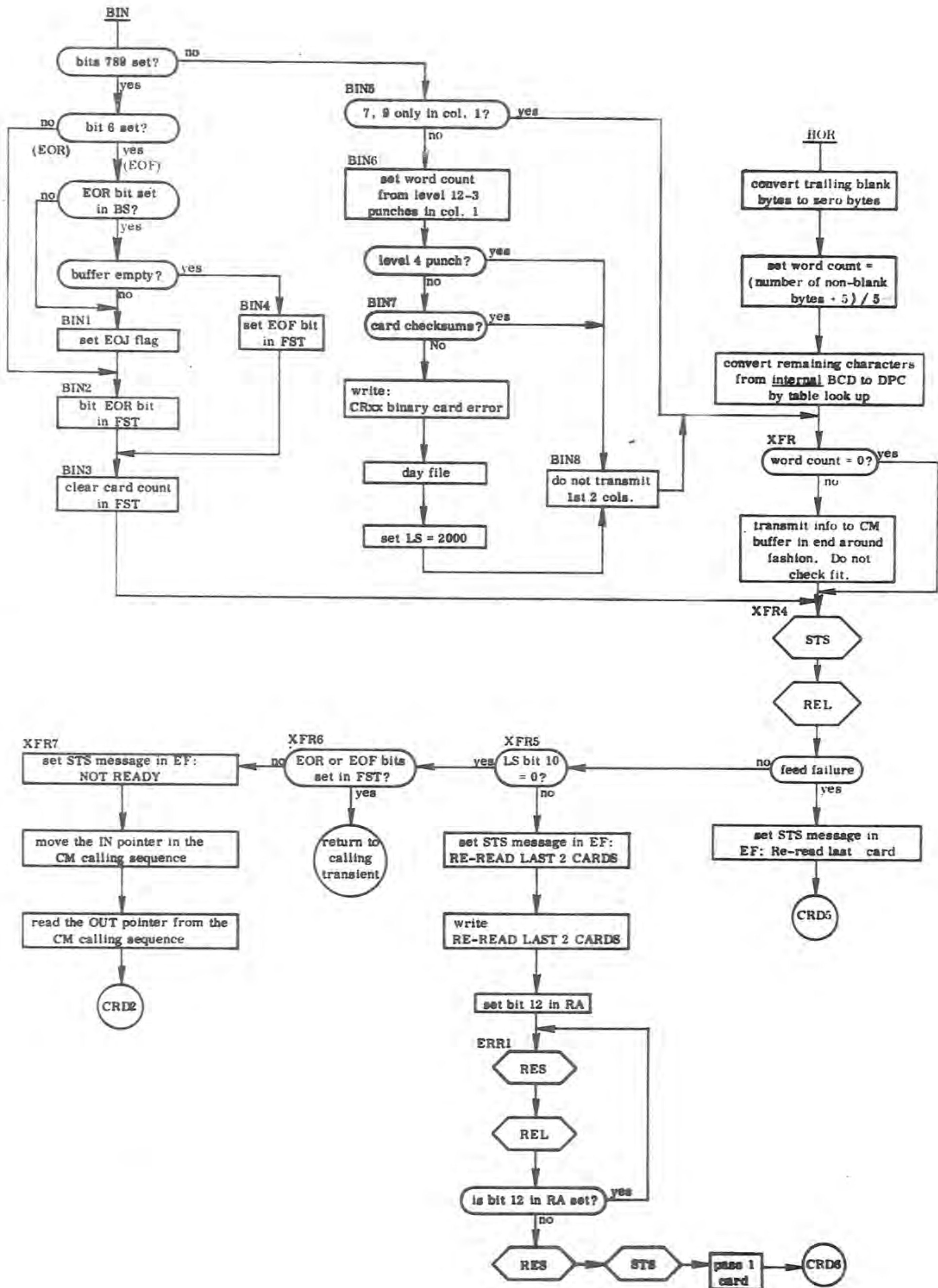
compare error
no
yes

write:
CRxx compare error

day file

CRD7
test status bit 2
set
BIN
reset
HOL





BIN
bits 789 set?
no
yes
BIN5
7, 9 only in col. 1?
yes
no
BIN6
set word count from level 12-3 punches in col. 1
level 4 punch?
BIN7
card checksums?
No
write: CRxx binary card error
day file
set LS = 2000
BIN8
do not transmit last 2 cols.
bit 6 set?
(EOR)
(EOF)
EOR bit set in BS?
buffer empty?
BIN4
set EOF bit in FST
BIN1
set EOJ flag
BIN2
bit EOR bit in FST
BIN3
clear card count in FST
HOR
convert trailing blank bytes to zero bytes
set word count = (number of non-blank bytes + 5) / 5
convert remaining characters from internal BCD to DPC by table look up
XFR
word count = 0?
transmit info to CM buffer in end around fashion. Do not check fit.
XFR4
STS
REL
feed failure
set STS message in EF: Re-read last card
CRD5
XFR5
LS bit 10 = 0?
set STS message in EF: RE-READ LAST 2 CARDS
write RE-READ LAST 2 CARDS
set bit 12 in RA
ERR1
RES
REL
is bit 12 in RA set?
RES
STS
pass 1 card
CRD6
XFR6
EOR or EOF bits set in FST?
return to calling transient
XFR7
set STS message in EF: NOT READY
move the IN pointer in the CM calling sequence
read the OUT pointer from the CM calling sequence
CRD2

convert trailing blank bytes to zero bytes

set word count = (number of non-blank bytes + 3) / 5

convert remaining characters from internal BCD to DPC by table look up

word count = 0?

transmit data to CM buffer in end around fashion (if not check it)

STS

REL

feed failure

set STS message in EF: Re-read last card

binary 789 only?

7, 9 only in col. 1?

set word count from level 12-3 punches in col. 1

level 4 punch?

card checksum?

write: CDxx binary card error

day file

set LS = 20000

do not transmit 1st 2 cols

buffer empty?

set EOY bit in FST

set CIU flag

EOR bit set in BUY

set card count in FST

EOR or EOY bits set in FST?

return to calling transient

set STS message in EF: NOT READY

move the IN pointer to the CM calling sequence

read the OUT pointer from the CM calling sequence

set STS message in EF: RE-READ LAST 2 CARDS

write RE-READ LAST 2 CARDS

set bit 12 in RA

RES

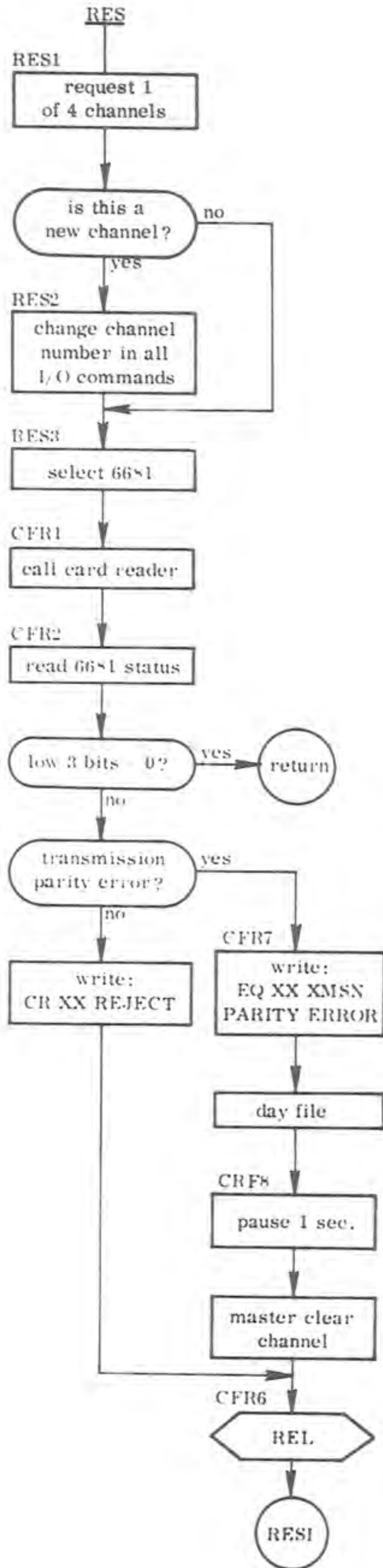
REL

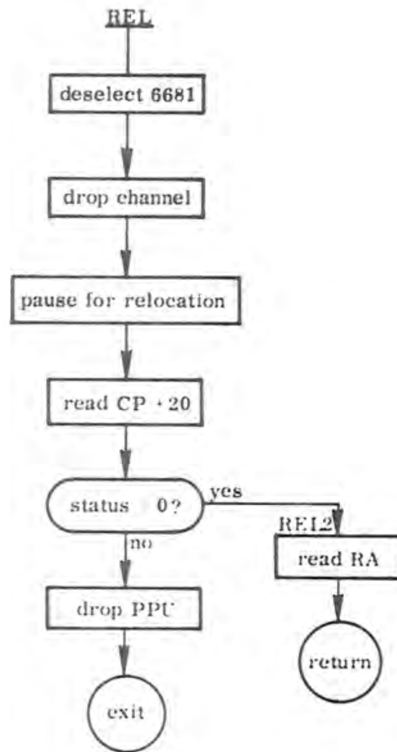
is bit 12 in RA set?

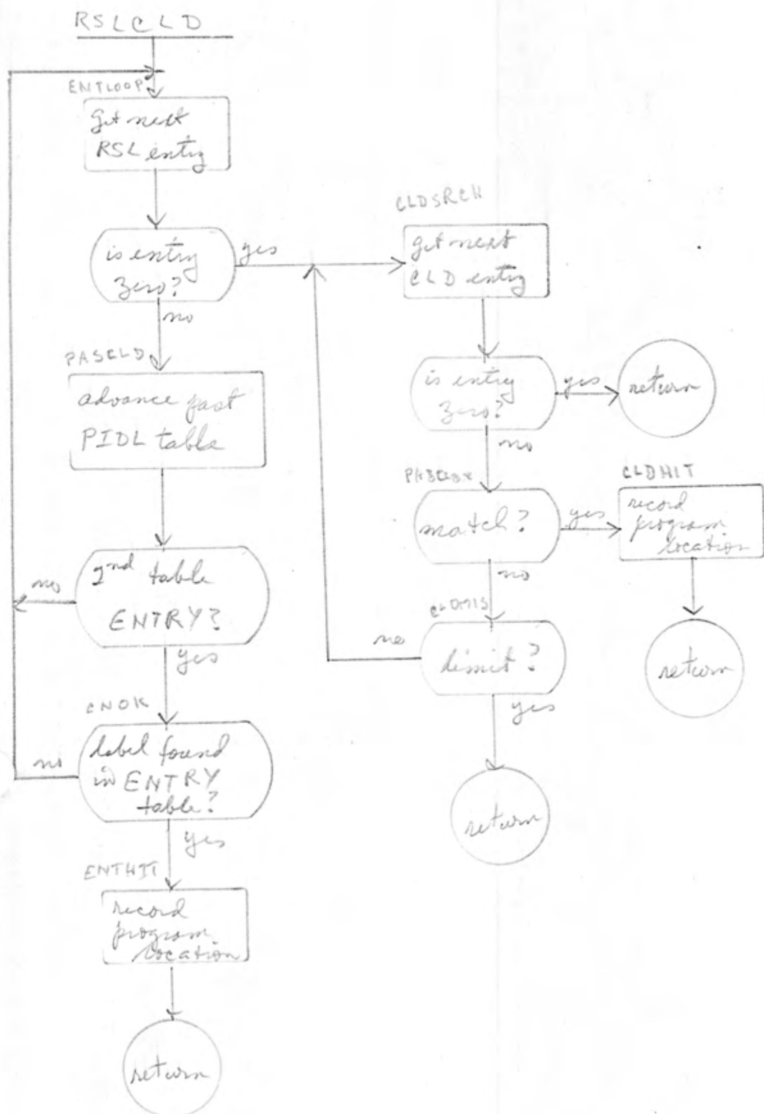
RES

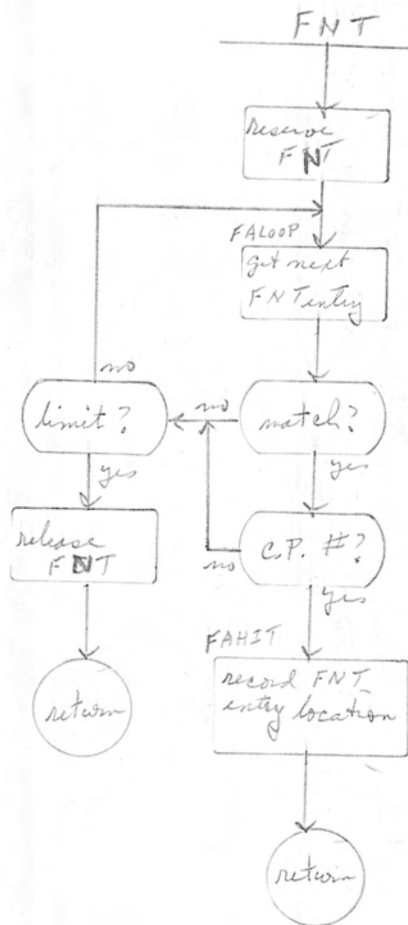
STS

pass 1 card


RES
RES1
request 1 of 4 channels
is this a new channel?
no
yes
RES2
change channel number in all I/O commands
RES3
select 6681
CFR1
call card reader
CFR2
read 6681 status
low 3 bits 0?
yes
return
no
transmission parity error?
yes
no
CFR7
write: EQ XX XMSN PARITY ERROR
write: CR XX REJECT
day file
CRF8
pause 1 sec.
master clear channel
CFR6
REL
RES1


REL
deselect 6681
drop channel
pause for relocation
read CP +20
status 0?
yes
REL2
read RA
no
drop PPU
return
exit


RSLCLD
ENTLOOP
Get next RSL entry
is entry zero?
yes
no
PASCLD
advance past PIDL table
2nd table ENTRY?
no
yes
CNOK
label found in ENTRY table?
no
yes
ENTHIT
record program location
return
CLDSRCH
get next CLD entry
is entry zero?
yes
return
no
match?
yes
CLDHIT
record program location
return
no
CLDMIS
limit?
no
yes
return

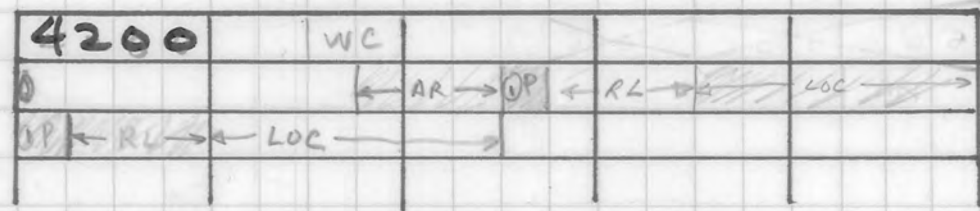

FNT
Reserve FNT
FALOOP
Get next FNT entry
match?
no
limit?
no
yes
release FNT
return
yes
C.P. #?
no
yes
FAHIT
record FNT entry location
return

PIDL

LCT
LOCAL common table

TEXT

FILL (common)



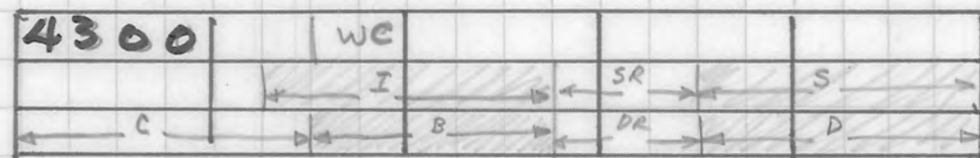
AR=0 Absolute
=1 Prog
=2 Neg
=3-77 LCT(AR=2)

R= 10 upper
= 01 middle
= 00 lower

LINK

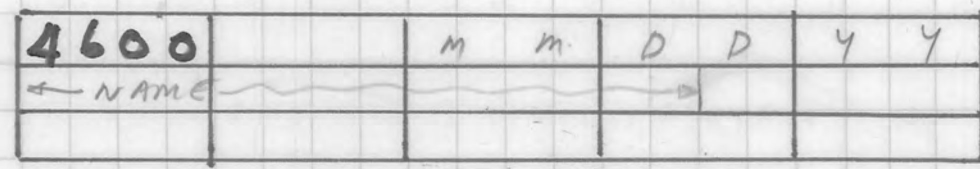
ENTR

REPL



B = Block size
I = Repetition Δ
C = # Repetitions

XFER



CARD FORMAT

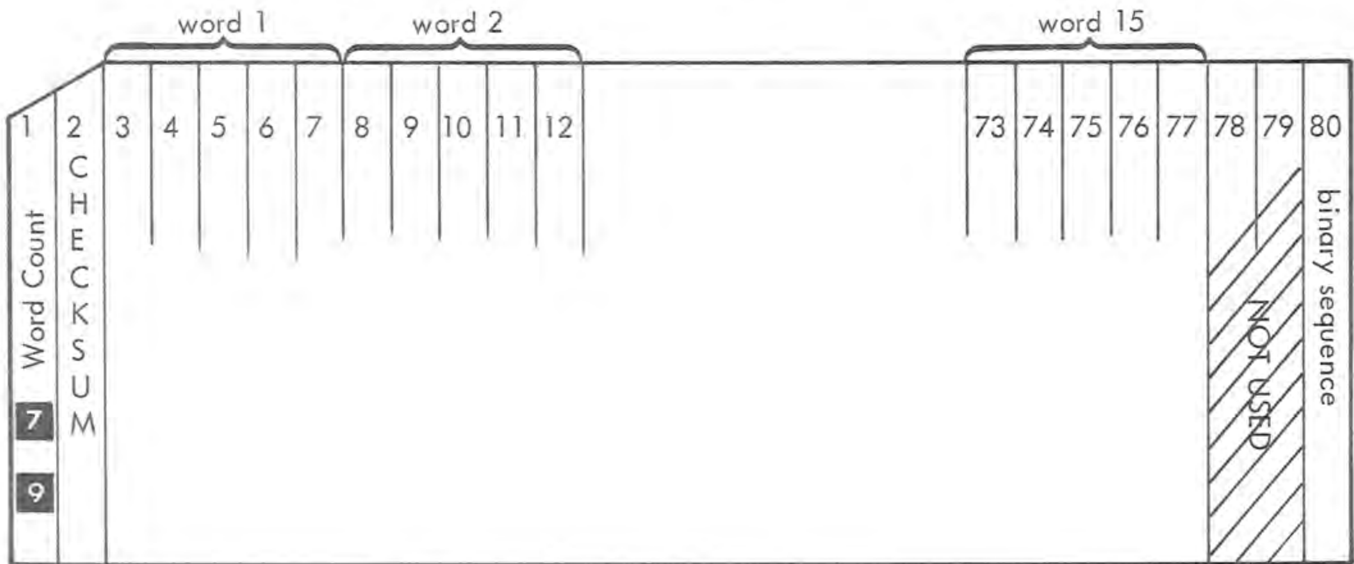
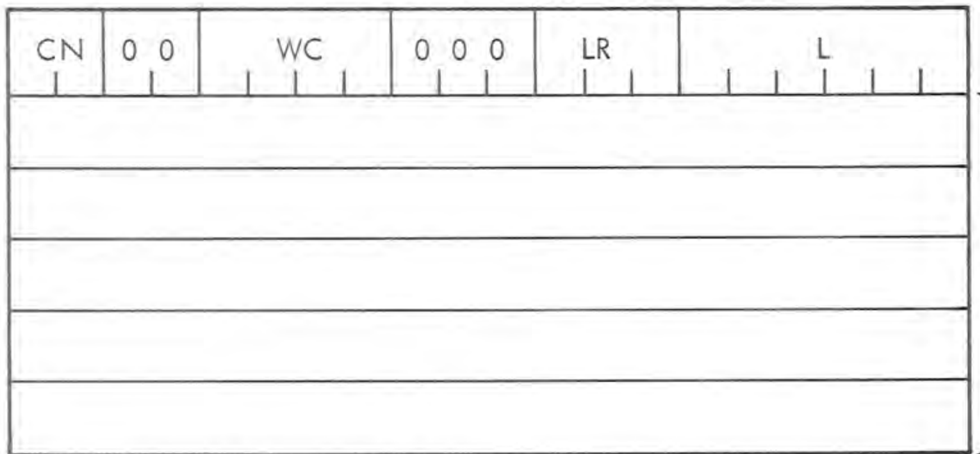


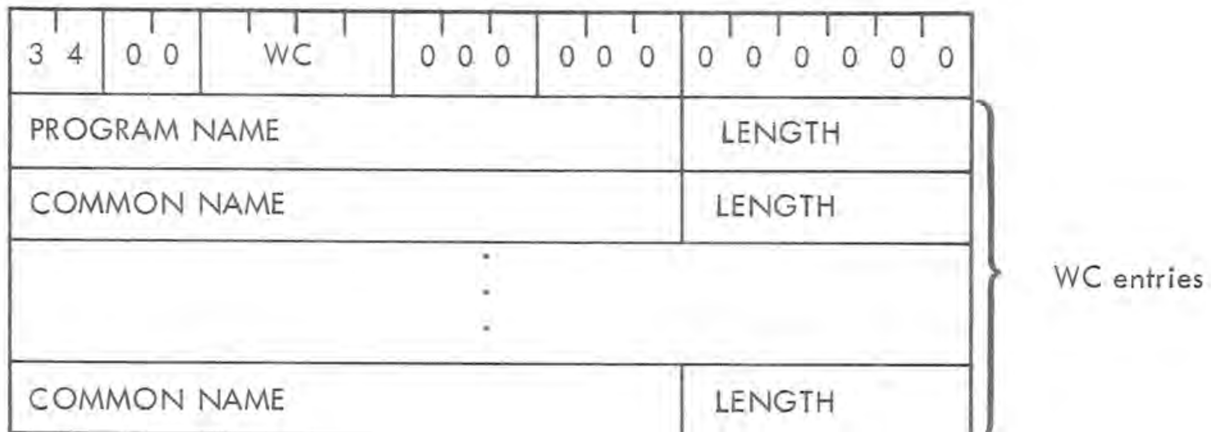
TABLE FORMAT



CN is Code Number: 34 = PIDL
40 = TEXT
42 = FILL
44 = LINK
36 = ENTR
43 = REPL
46 = XFER
77 = label
50 = absolute overlay

LR and L fields are zero except in the TEXT tables.

PIDL

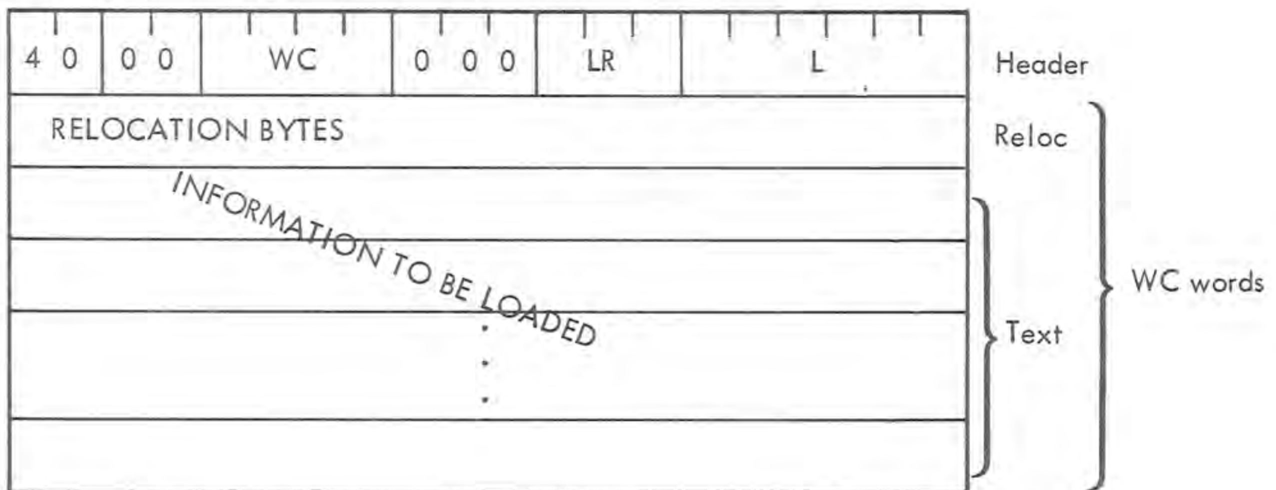


The first entry is always the program name. Subsequent entries are the names of common blocks.

Blank common is treated just like any other Common block, but the name is left blank (55555555555555).

All names are left justified zero filled.

TEXT



WC is limited to 16_{10} or less. Where there is more test, extra (not longer) TEXT tables are required.

LR specifies the area into which the test is to be loaded, as follows:

0 = absolute (relative to RA)
1 = program (relative to 1st location of program)
2 illegal
3 = 1st common (relative to 1st common block mentioned in PIDL table)
⋮
n = n-2th common (relative to n-2th common block mentioned in PIDL table)

PIDL

[illegible]	[illegible]
LENGTH	[illegible]

$N \geq$ entries

[illegible] ... are the names of common [illegible]

[illegible] ... and the name of the table [illegible]

Header

Word count

TEXT

[illegible] ... (entry). When ... longest TEXT tables are [illegible]

[illegible] ... into the text to be loaded as follows:

[illegible] ... relative to XA,
[illegible] ... relative to 1st location of program)

[illegible] ... (relative to 1st common block mentioned in PIDL table)

[illegible] ... n-2th common (relative to n-2th common block mentioned in PIDL table)

L is the address (within specified area) into which the first word of text is to be loaded. Subsequent words go into the succeeding locations.

(Encountering an ORG card or a BSS card during assembly causes the punching of a short TEXT table.)

The relocation bytes are 4 bits apiece. 15 (or WC-1 if WC < 16) are kept in the relocation word. Because only 15 relocation bytes can be kept in the relocation word, the TEXT tables are limited to 16 words each. Each TEXT table has exactly one relocation word — it is always the word immediately following the header.

Only 30 bit instructions have relocatable addresses, and these instructions must therefore occupy either an upper, middle, or lower position in the text word. Therefore the relocation bytes need merely concern themselves with the upper, middle, and lower position of a word, not with the 4 positions associated with parcels. Furthermore, if a word carries a middle instruction, it cannot carry an upper or lower instruction. (It can carry 2 more instructions but these are 15 bits and associated with Parcels 0 and 3, not upper and lower.) The relocation bytes in the TEXT table are only concerned with positive and negative PROGRAM relocation.

upper bit set = upper relocation

second bit: 0 = positive
1 = negative

second bit set = middle relocation (assuming not upper!)

third bit: 0 = positive
1 - negative

third bit set = lower relocation (assuming not middle!)

fourth bit: 0 = positive
1 = negative

1000 = upper +
1100 = upper -
1010 = upper + lower +
1011 = upper + lower -
1110 = upper - lower +
1111 = upper - lower -
0010 = lower +
0011 = lower -
0100 = middle +
0110 = middle -
0000 = no relocation

UNUSED

0001 = no relocation
0101 = middle +
0111 = middle -
1001 = upper +
1101 = upper -

L is the address (within specified area) into which the first word of text is to be loaded. Subsequent words go into the succeeding [illegible].

Encountering [illegible] card or [illegible] 855 [illegible] causes the punching of a [illegible] TEXT table.

The relocation [illegible] is between [illegible] words [illegible] the text location word. Between [illegible] relocation word [illegible] relocation word, the TEXT tables are limited [illegible] TEXT [illegible] relocation word — it is always the word [illegible] following the [illegible].

Only [illegible] instructions have relocatable [illegible] instructions must therefore [illegible] Therefore the relocation [illegible] the upper, middle, and lower positions of a word, [illegible] If a word contains middle [illegible] instructions [illegible] upper and lower). The relocation [illegible] PROGRAM relocation.

[illegible] upper relocation

[illegible] bit [illegible] 0 = positive
[illegible] 1 = negative

[illegible] middle relocation [illegible]

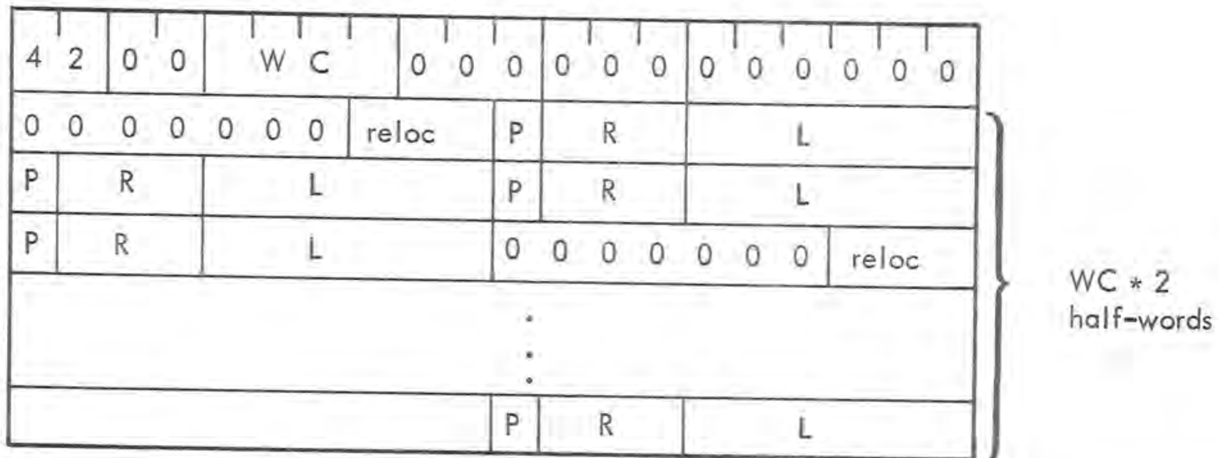
[illegible] 0 = positive
[illegible] 1 = negative

[illegible] lower relocation [illegible]

[illegible] 0 = positive
[illegible] 1 = negative

			UNUSED
[illegible]	upper		
[illegible]	upper		
[illegible]	upper + lower	[illegible]	relocation
[illegible]	upper + lower	[illegible]	middle
[illegible]	upper + lower	[illegible]	middle
[illegible]	upper + lower	[illegible]	upper
[illegible]	lower	[illegible]	upper
[illegible]	lower		
[illegible]	middle		
[illegible]	middle		
0000	no relocation		

FILL



The fill table consists of 2 * WC half-words ordered as a control half-word followed by a string of detail half-words, another control half-word, and another string of detail half-words, etc. filling out the table. The last half-word may be all zeros if unused.

Control half-words have a zero upper bit and detail half-words have their upper bit set.

The control half-words only specify the relocation factor as follows:

0 = absolute
1 = program
2 = negative program
3 = 1st common
.
.
.
n = n-2th common

This relocation factor is added to address fields as specified by the detail half-word as follows:

R = region: 0 = absolute
1 = program
2 = illegal
3 = 1st common
.
.
.
n = n-2th common

L = address within that region

P = position within that location:
100 = lower
101 = middle
110 = upper
} high bit always set in detail half-words.
111 not used

EXAMPLE

SOURCE

```
         IDENT   PROG
         ORG     2
P1       VFD     D30/INPUT,N12/0,A18/BE1
P2       VFD     D36/OUTPUT,N6/0,A18/BE2
P3       VFD     D42/SCRATCH,A18/BE3
ALPHA    COMMON  AL1,2000B,AL2,2000B
BETA     COMMON  BE1,5,BE2,5,BE3,5
         COMMON  BUFFER,2000B
         ORG     *
         SA1     P1-2
         SB1     A1
         MX3     42
LOOP     SA1     A1+B1
         ZR      X1,OUT
         BX6     X1*X3
         SA6     X1
         ZR      LOOP
         ORG     BE1
         VFD     D30/INPUT,N30/0
         CON     AL1
         CON     AL1
         CON     AL1
         CON     AL2
         VFD     D36/OUTPUT,N24/0
         CON     AL2
         CON     AL2
         CON     AL2
         CON     AL2+2000B
         VFD     D42/SCRATCH,N18/0
         CON     BUFFER
         CON     BUFFER
         CON     BUFFER
         CON     BUFFER+2000B
         ORG     *
OUT
```

OBJECT

Object word	
34000004000000000000	PIDL
20221707000000length	
01142010010000004000	
02052401000000000017	
55555555555555002000	
40000004000000000002	TEXT ORG 2
00000000000000000000	
11162025240000000000	
17252420252400000005	
23032201240310000012	
40000004000001000000	TEXT ORG *
01040000000000000000	
51100000016411043352	
54111030100000311613	
53610040000000146000	
40000020000003000000	TEXT ORG BE1
00000000000000000000	
11162025240000000000	
00000000000000000000	
00000000000000000000	
00000000000000000000	
00000000000000002000	
17252420252400000000	
00000000000000002000	
00000000000000002000	
00000000000000002000	
00000000000000004000	
23032201240310000000	
00000000000000000000	
00000000000000000000	
00000000000000000000	
00000000000000002000	
42000012000000000000	FILL
00000000006001000000	
00000000034004000001	
40040000024004000003	
40040000044004000006	
40040000074004000010	
40040000110000000004	
40000000024000000003	
40000000040000000005	
40040000134004000014	
40040000154004000016	
4000xxxx000001000003	TEXT ORG *

LINK

4 4 0 0	W C		0 0 0	0 0 0	0 0 0 0 0 0	
name of external					0 0 0 0 0 0	WC words
P	R	L	P	R	L	
P	R	L	name of			
external		0 0 0 0 0 0	P	R	L	
. . .						
			P	R	L	

The LINK table consists of a control word followed by a string of detail half-words, then a second control word (it may be divided between two CM words as shown) and another string of detail half-words etc. filling out the table. The last half word may be all zeros if unused.

The control words merely contain the external symbol in display code, left justified and zero filled. Therefore, the uppermost bit is reset. The upper bit in the detail half-words is always set.

The detail half-words following a particular control word specify the address fields in which a reference to the particular external was made. They have the same form as in the FILL table. Namely,

P = position	R = region	L = location
4 = lower	0 = absolute	
5 = middle	1 = program	
6 = upper	2 illegal	
7 not used	3 = lst common	
	.	
	.	
	.	
	n = n-2th common	

After loading is complete, the external symbols will have entry locations, and these addresses are plugged into the control word by LOADER. Then the address from the control word is added to the address fields specified by the detail half-words completing the program linkage. Notice that this treatment of program linkage allows external arithmetic in address fields at compile time.

ENTR

3 6	0 0	W C	0 0 0	0 0 0	0 0 0 0 0 0
name of entry point					0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0				R	L
name of entry point					0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0				R	L
. . .					
entry point symbol					0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0				R	L

(All rows after the first: WC/2 entries)

Each entry point requires two words in the ENTR table. The first word consists of the entry point symbol left justified and zero filled. The second word contains the entry address. The R and L fields are as before. Note entry points are allowed in common areas!

EXAMPLE

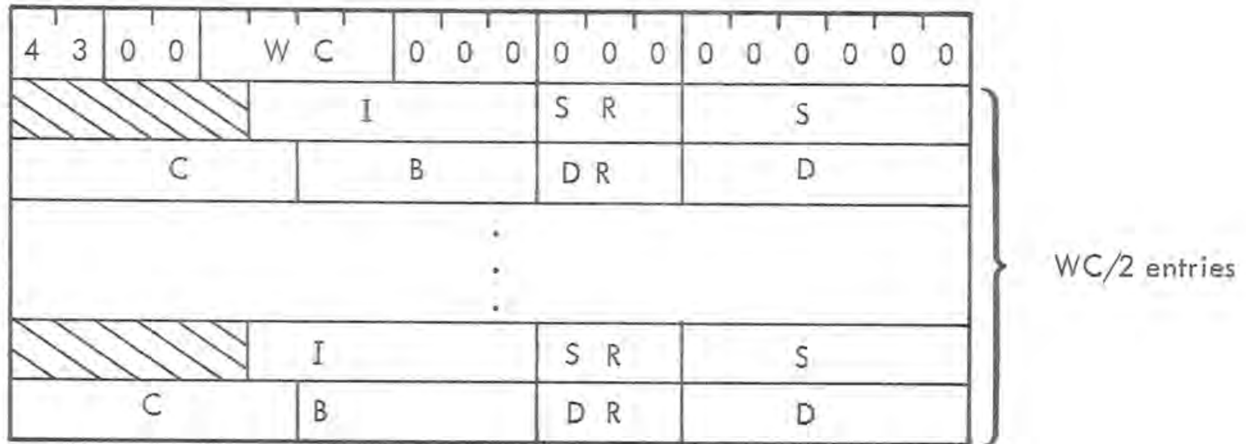
SOURCE

```
        IDENT  GREEK
        ENTRY  ALPHA,BETA,GAMMA
        EXT    ALEPH,BETH,GIMMEL
ALPHA   CON    0
        RJ     ALEPH
DELTA   SX6    B7
        SA6    A0
        RJ     GIMMEL
        SA1    A0
        JP     ALPHA
BETA    CON    0
        SA1    BETA
        BX6    X1
        SA6    ALPHA
        RJ     BETH
        JP     DELTA
GAMMA   CON    0
        SA1    GAMMA
        BX6    X1
        SA6    GIMMEL
        JP     GIMMEL+1
```

OBJECT

Word	Table
34000001000000000000	PIDL
07220505130000length	
36000006000000000000	ENTR
01142010010000000000	
00000000000001000000	
02052401000000000000	
00000000000001000004	
07011515010000000000	
00000000000001000010	
44000006000000000000	LINK
01140520100000000000	
60010000010205241000	
00000000004001000006	
07111515051400000000	
40010000026001000012	
40010000120000000000	
4000xxxx000001000000	TEXT
00002010420040xxxxxx	
00000000000000000000	
01000000004600046000	
etc.	

REPL



Each entry in the REPL table is two words. The fields are as follows:

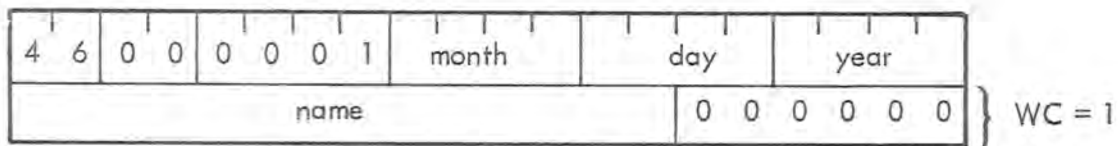
SR	source area
S	source address (within the specified area)
B	block size (number of words to be replicated) (0 is interpreted as 1)
DR	destination area
D	destination address (within that region) (0 is interpreted as S + B)
C	number of replications (0 is interpreted as 1)
I	replication interval (0 is interpreted as B)

For example if the count were 3, the information from S through S+B-1 would be reproduced at D through D+B-1, at D+I through D+I+B-1, and at D+2I through D+2I+B-1.

DR and SR fields are as follows:

0 = absolute
1 = program area
2 illegal
3 = lst common area
⋮
n = n-2th common area

XFER



The XFER table always contains a word count of 1. The month, day and year fields in the header are optional and ignored by GPSL; if present they are in display code. The XFER entry is left-justified, zero-filled. It must be a declared entry point to some program, but not necessarily the current one. Its address field is filled in by LOADER just before execution.

Label

7 7	0 0	0 0 1 6	0 0 0	0 0 0	0 0 0 0 0 0
program name					0 0 0 0 0 0
0 0	0 0	0 0 0 0	0 0 0	0 0 0	0 0 0 0 0 0
. . .					
0 0	0 0	0 0 0 0	0 0 0	0 0 0	0 0 0 0 0 0

WC = 14

The label table always has a word count of 14_{10} = 1 card exactly. GPSL ignores the label table — its only use is as a header block for routines such as COSY and COPYN.

ASCENT will always punch the label card at the head of the object deck. Because it is an entire card by itself, and ignored by GPSL, the programmer may remove the card or not, as he desires, before loading.

Absolute Overlay

5 0	0 0	L 1	L 2	Load Address	Entry Address
. . . .					

Words 2 through end of logical record are absolute text.

The first word of an absolute overlay identifies it and distinguishes this record from other records or tables GPSL may encounter. The format of the first word contains no word count, and an absolute overlay is not really a table in the sense used above.

LOADER TABLES

Pointers

- υ points to the beginning of the first LINK table relative to the PTBL table for program 1
- φ points to the beginning of the first ENTR table
- χ points to the beginning of the first FILL table
- ψ points to the beginning of the first REPL table
- ω points to the next PTBL table

If any of these tables should be absent, the corresponding pointer is zero. All pointers are 12 bits.

If there is no succeeding PTBL table the ω pointer is also zero.

Υ, Φ, X, Ψ, Ω have the same function for PROGRAM 2, and their values point to the corresponding tables relative to the PTBL table for program 2.

Blank common starts at the end of last loaded program and may overlay the LOADER and its tables.

During Segment or Overlay loading, blank common is appended to the end of whatever segment or overlay defines it . In the case of Overlay loading, blank common may overlay the LOADER and its tables only if it is defined in the lowest level overlay, or subsequent overlay control cards use the C-field to restrict blank common length. In the case of Segment loading, blank common may not overlay the LOADER.

RA	
RA+100	SECTION Table
	SEGMENT Table
	Common
	PROGRAM 1
	Common
	PROGRAM 2
Blank Common	Unused Core
	0
	REPL Tables FILL Tables ENTR Tables LINK Tables
	⋮
	Common name address
	Υ Φ X Ψ Ω
	PROGRAM 2 name address
	0
	REPL Tables FILL Tables ENTR Tables LINK Tables
	⋮
	Common name address
	υ φ χ ψ ω
	PROGRAM 1 name address
	previous XFER address
	blank common pointer
	XFER name address
	LOADER
FL	

ENTR — Table has same format as ENTR table from object file. LOADER supplies the absolute address

Next Table	
reloc.	rel.loc.
3rd entry	abs. loc.
reloc.	rel. loc.
2nd entry	abs. loc.
reloc.	rel. loc.
1st entry	abs. loc.
0	pointer to next ENTR table

LINK — Table has same format as LINK table from Object file. LOADER supplies the absolute address.

Next Table	
nal abs.loc.	P R rel. loc.
P R rel. loc.	2nd exter-
P R rel. loc.	P R rel. loc.
1st External	abs. loc.
0	pointer to next LINK table

FILL — Table has same format as FILL table from Object file.

Next Table	
P R rel. loc.	P R rel. loc.
P R rel. loc.	control byte
control byte	P R rel. loc.
0	pointer to next FILL table

REPL — Table has same format as REPL table from Object file.

Next Table			
C	B	DR	D
	I	SR	S
C	B	DR	D
	I	SR	S
0	pointer to next REPL table		

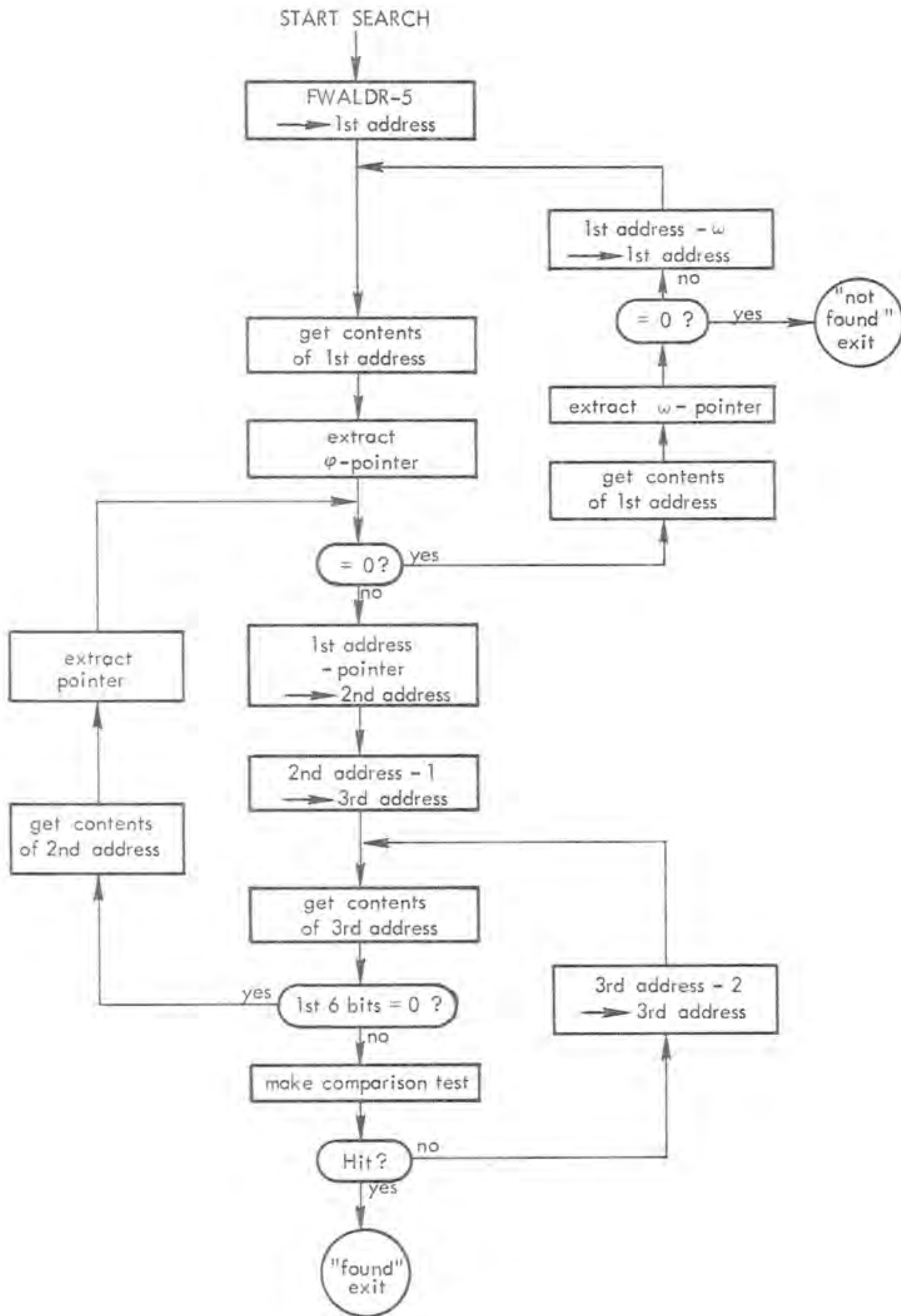
ENTR Table has same format as ENTR table from object files. LOADER supplies the [illegible] address.

LINK Table has same format as LINK table from Object File. LOADER supplies the absolute address.

FILL Table has same format as FILL table from Object File.

REPL Table has same format as REPL table from Object File.

THREADED LIST SEARCH



	. . .
	0 ptr
	n7
	n6
E	0 p3
	. . .
D	φ2=D-E ω2
	Prog 2
	0
	. . .
	0 ptr
	n5
	n4
C	0 p2=0
	. . .
	0 ptr
	n3
	n2
	n1
B	0 p1=A-C
	. . .
A	φ1=A-B ω1=A-D
	Prog 1
	X
	BC
	X
	LOADER

LOADER ACTION FOR OVERLAY GENERATION

1. Complete present overlay (LINK tables processed and library routines loaded).
2. Fill unsatisfied externals with out-of-bounds references.
3. Establish or link blank common.
4. Set LWA overlay from CORNEXT in proper word of OVLINPT Table. Also set TBLNEXT into proper file-name entry of OVLINPT Table.
5. Return to LDR unless Write-Overlay bit or End-of-Load bit is set.

If either bit is set:

6. Scan the file name entries to find the most principal level not yet written.
7. Make up a 77 table entry at CORNEXT+1 through CORNEXT+17_8.
8. Make up a CIO call using file name specified for a binary write.

   Set: FIRST = FWA overlay (from header)

   IN = LWA overlay + 1 (from one of last six words in OVLINPT Table)

   OUT = CORNEXT+1 (beginning of 77 table)

   LAST = CORNEXT+20_8 (end of 77 table +1)
9. Call CIO.
10. When write is complete, set high bit in corresponding file name entry.
11. Repeat steps 6 — 10 until all overlays in core have been written.
12. Reset CORNEXT and TBLNEXT for generation of next overlay (whose level is already known).
13. Return to LDR unless End-of-Load bit is set.

If the End-of-Load bit is set:

14. If a LOAD card had been processed, place END in RA + 1.

    MTR will call 1AJ
15. If a program call card, LOADER calls LDR to load the initial 0,0 overlay.

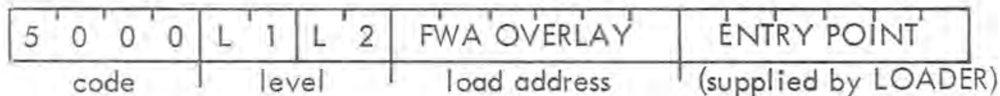
    LDR will enter the user program and not return to LOADER.

OVLINPT TABLE (Located within LOADER)

Location	Contents	Label
FWALDR + 256_8	INITIAL FILE NAME	OVLINPT
OVLINPT + 1	FIRST WORD OF OVERLAY 0,0 HEADER	OVLZZHD
+ 2	OVERLAY 0,0 FILE NAME	OVLZZFN
+ 3	FIRST WORD OF OVERLAY n,0 HEADER	OVLPRHD
+ 4	OVERLAY n,0 FILE NAME	OVLPRFN
+ 5	FIRST WORD OF OVERLAY n,m HEADER	OVLSCHD
+ 6	OVERLAY n,m FILE NAME	OVLSCFN
+ 7	LAST OUTPUT FILE NAME	LASTOVF
+ 8	INPUT FILE NAME	OVLINFN
+ 9	OVERLAY 0,0 BLANK COMMON POINTER	OVLZBCM
+ 10	OVERLAY 0,0 LAST WORD ADDRESS + 1	OVLZLWA
+ 11	OVERLAY n,0 BLANK COMMON POINTER	
+ 12	OVERLAY n,0 LAST WORD ADDRESS + 1	
+ 13	OVERLAY n,m BLANK COMMON POINTER	
+ 14	OVERLAY n,m LAST WORD ADDRESS +1	

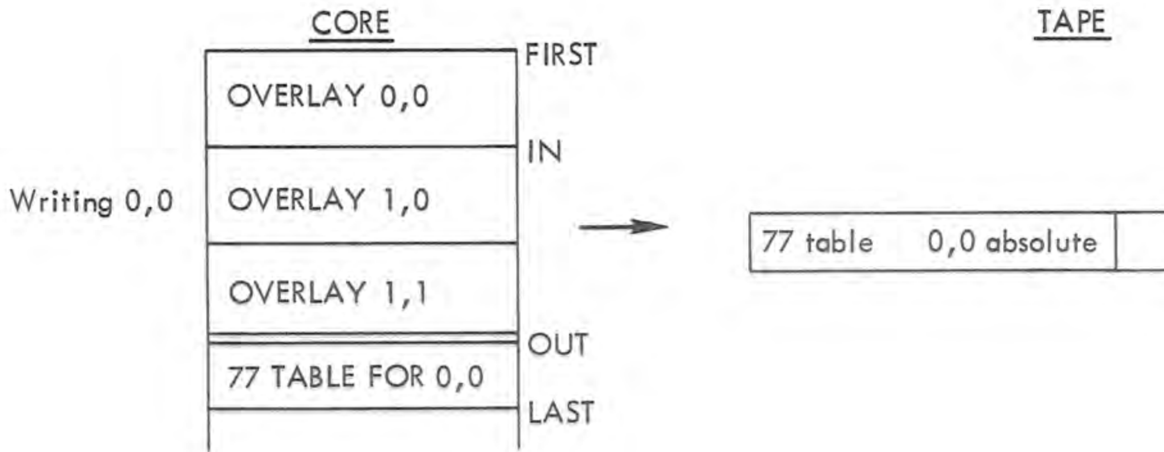
Contained in the low order 18 bits of locations OVLZZFN, OVLPRFN and OVLSCFN is the first word address of the Loader tables for that overlay (= TBLNEXT for subordinate overlay).

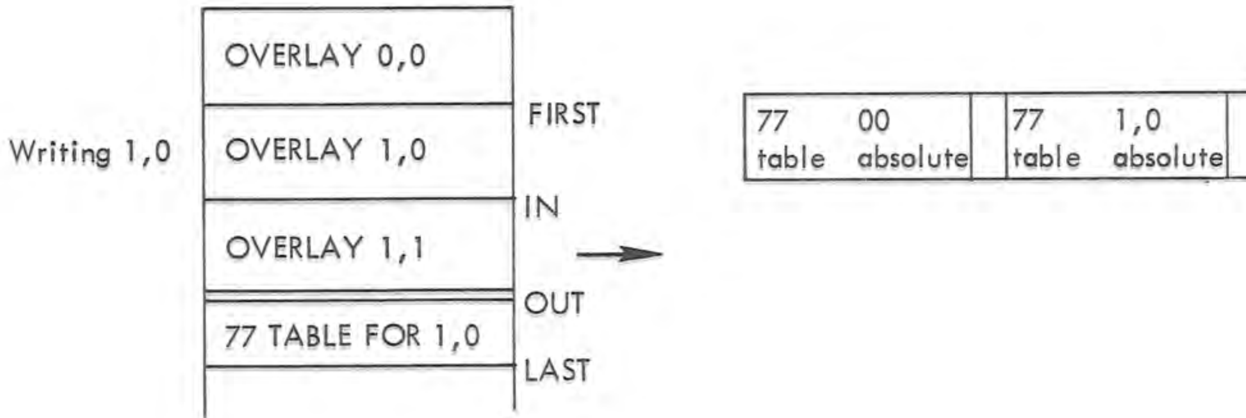
In the header words — OVLZZHD, OVLPRHD and OVLSCHD, the format of the information is:

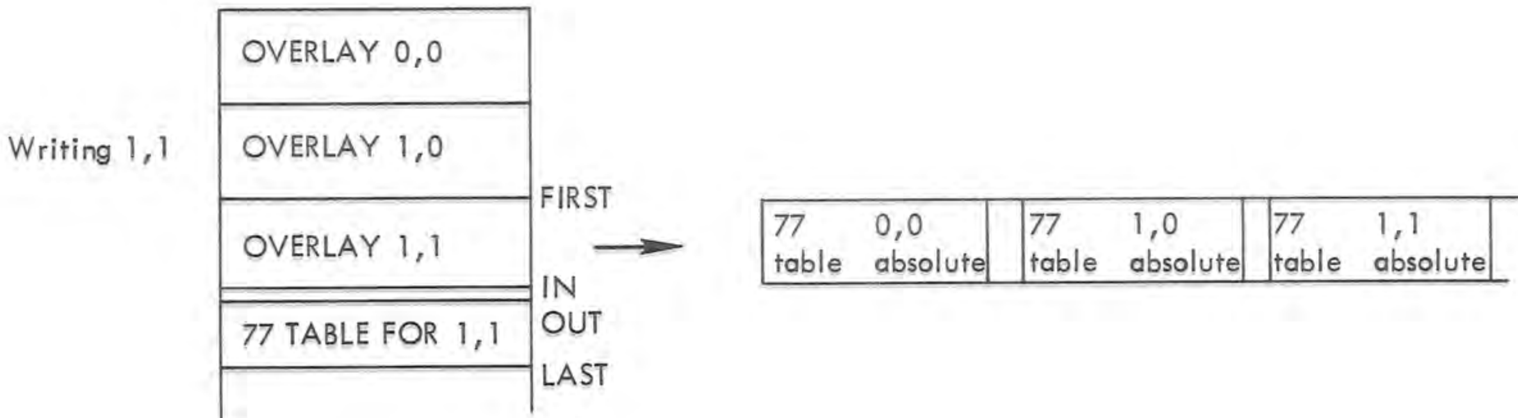


When the particular overlay is written out on its absolute file, the high bit in the corresponding file name (OVLZZFN, OVLPRFN or OVLSCFN) is set. In this way, LOADER is aware of which overlays have been written out, and which have yet to be written. When a new overlay is generated, the corresponding file name is rewritten (see program 2LA routine OVERLAY) and therefore the high bit is reset, therefore LOADER knows that the absolute text has not yet been written in the absolute overlay file.

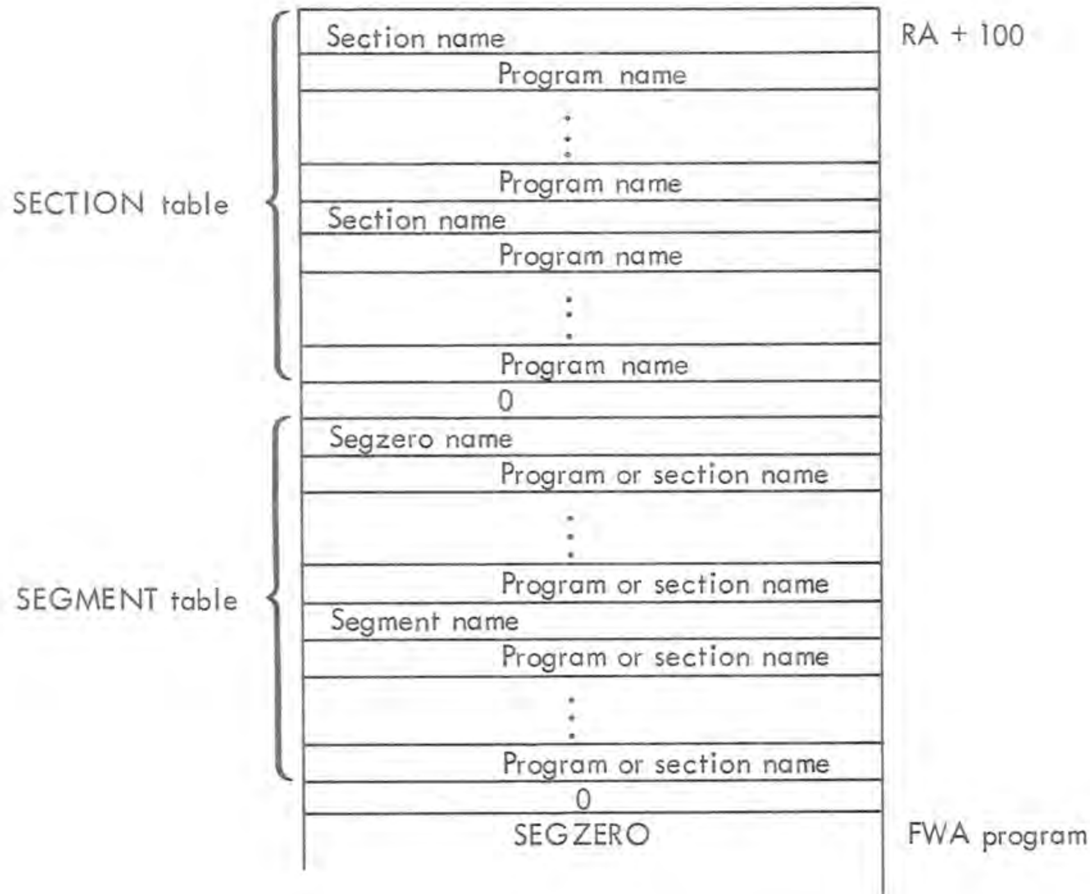
EXAMPLE







SEGMENT and SECTION tables



To distinguish section names from program names in the SECTION table, the high order bit in the section names is set. Similarly in the SEGMENT table the high order bit of each segment name entry is set. All names are inserted left-justified and zero-filled. The end of each table is indicated by a zero word. The SECTION table, if there is one, always starts at RA + 100. The address of the beginning of the SEGMENT table is kept in the β-field of RA + 65. The FWA for the program is kept in the low order 18 bits of RA + 66.

A section may consist only of a string of programs that are to be loaded from a file other than the system library. All the programs within a section must be on the same file. All sections must be defined before the first segment.

A Segment may consist of a string of programs (that are to be loaded from a file other than the system library) and previously defined sections. All the programs within a segment as defined here must be on the same file. (In a later definition of Segment this restriction is removed.) In particular, all the programs constituting SEGZERO (in either definition of segment) must lie on the INPUT file.

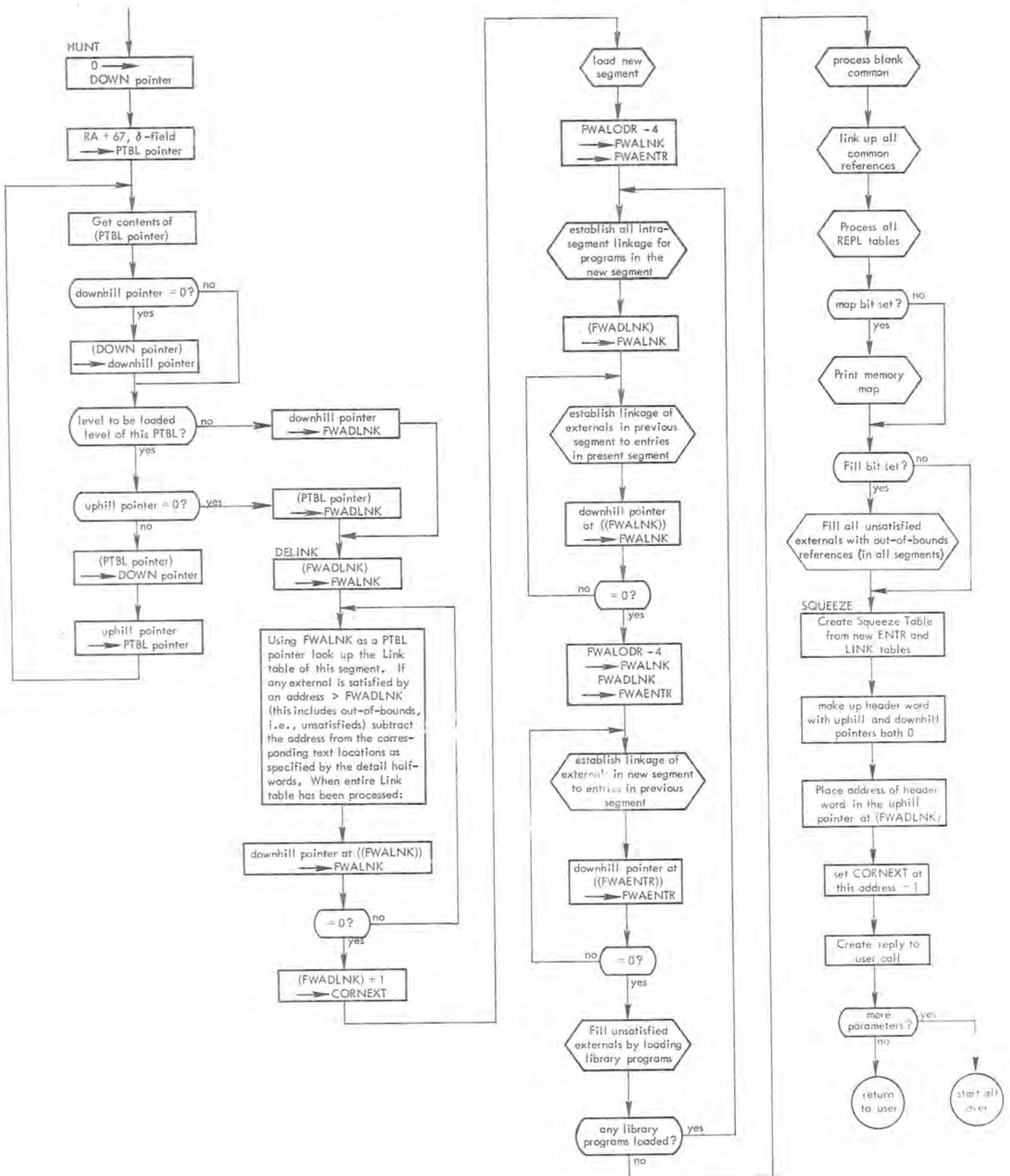
SQUEEZE Tables

Label	Contents			Note
	SECTION Table			RA + 100
	0			
	SEGMENT Table			
	SEGZERO			
	SEGZERO SQUEEZE Table			
	Newly Loaded SEGMENT	Prog 1		
		Prog 2		
	0			
	e1	abs address		
	e2	abs address		
	e3	abs address		
	e4	abs address		
	e5	abs address		
	e6	abs address		
T	0			
	c2	c1		
	L1	c3		
	c4	L1		
	c6	c5		
	L2			
	L3	c7		
	c8	L3		
	PPPPPP	c9		
	L4			
	0			
S	$\nu=1$ $\varphi=S-T$ $\chi=0$ $\psi=0$ $\omega=0$			
	level	downhill	uphill	
	⋮			

Uphill and downhill pointers are zero at time of creation. They are set later during segment linking to the next level segment. The pointer to the SEGZERO PTBL is located in RA + 67. The other PTBL's are found by threading.

Label	Contents			Note
	⋮			
	0		ptr	
	e6			
	e5			
H	0		p3	ENTR 1 (Prog 2)
	⋮			
	0		ptr	
	c8	c9		
	L4			
G	0		$\upsilon3$	LINK 1 (Prog 2)
	⋮			
F	$\Upsilon=F-G$	$\Phi=F-H$	Ω	PTBL 2
	Prog 2			
	0			
	c7	0		
	L3			
E	0		$\upsilon2 = 0$	LINK 2 (Prog 1)
	⋮			
	0		ptr	
	e4			
	e3			
D	0		p2 = 0	ENTR 2 (Prog 1)
	⋮			
	0		ptr	
	c5	c6		
	L2	c4		
	c3	L2		
	c1	c2		
	L1			
C	0		$\upsilon1=A-E$	LINK1 (Prog 1)
	e2			
	e1			
B			p1=A-D	ENTR 1 (Prog 1)
	⋮			
A	$\upsilon=A-C$	$\varphi=A-B$	$\omega=A-F$	PTBL 1
	Prog 1			
	X			
	BC			
	X			
	LOADER			

SEGMENT LINKING – GENERALIZED



CHAPTER 13 - SYSTEM PERIPHERAL PROCESSOR ROUTINES

(TABLE OF CONTENTS)

ACE	Control Card Reader	13-2
APR	Automatic Program Sequencer	13-4
CHK	Check Output File	13-12
CTS	COMMON File Processing	13-13
DMP	Dump CM	13-16
LDV	Absolute Overlay Loader	13-28
LOC	Load Octal Cards	13-30
MDI	Move System Directory (EDITLIB)	13-31
MEM	Process MEMORY Function	13-36
MSG	Add Message to Dayfile	13-155
REQ	REQUEST Card/Function Processing	13-37
RFL	Request Field Length	13-42
SRB	Enter Expanded Disk Address into Directory (EDITLIB)	13-43
XDQ	PP Counterpart of XXXDMPQ	13-46
XRQ	PP Counterpart of XXXRESQ	13-48
1BT	Blank Label Tape and Disk	13-49
1DF	Dump Dayfile	13-50
1MF	Multifile Positioning	13-51
1MR	OPEN Read/Alter Magnetic Tape File	13-52
1MW	OPEN Write Magnetic Tape File	13-53
1PL	Dummy Plot Program	13-54
1RI	Roll-In	13-55
1RO	Roll-Out	13-58
1TD	Tape Dump	13-60
2BP	Check Buffer Parameters	13-62
2DF	Drop File(s)	13-64
2LP	On-Line Printer Driver	13-65
2PC	On-Line Punch Driver	13-88
2RC	On-Line Card Read Driver	13-105
2TJ	Translate Job Card	13-115
3RP	Close Reel Processing	13-117
5DA	Private Disk Packs	13-123
6PC	Drop Permanent Mass Storage Files	13-154

ACE CONTROL CARD READER

Design Philosophy

ACE was patterened after the control card reading routines in 1AJ. ACE was designed for use by a central program processing multiple card control statements. The function codes are based on those of CIO.

Logic Flow

ACE, when loaded, checks a table for the validness of the function code and the subroutine to use. Each function is processed by that subroutine and the use of various housekeeping routines.

Major Subroutines

SETFNT — Set up the FNT for READ or SKIP
The second word of the Input FNT is saved along with the last code/status from the third word. The function code is inserted in the FNT and the FST for the control cards is obtained from the control point area.

REFNT — Reset the FNT
The old code/status is returned to the FNT along with the FST. The control card FST is obtained from FSTCC (D.EST) in the PP, and returned to the control point area.

TRANS — Transfer data
The amount of data read is computed and it is transferred to the buffer in the control point area. The number of PP bytes read is in D.T3+C.RWPPLW and the CM word count is returned in WORDS.

Function Processing

READ — 10B
The routine READ is used. The control card is moved one word at a time to R.A+70B to RA+77B. Zero words fill from the end of the control card, denoted by a zero in the last byte, to RA+100B. If the card is split accross PRU's, RCCF is used to read the next PRU. If the first word of the card is zero, the end-of-record status (bit 4 of status word) is set meaning end of control cards.

BACKSPACE — 40B
Since the end of a card is denoted by a zero in the last byte, this routine will read backward, a word at a time, until the second zero fifth byte is found. The "next statement pointer" is placed at the following word which is the start of the previous control card. RCCB is called if the top of the buffer is reached. If RCCB finds the present buffer load is the first PRU, the pointer is left at the start of the buffer. When the backspace is issued to back to the start of the first card or beyond. The end-of-record is set.

Environment

No other routines are used. Stack requests are put through both R.READP and R.EREQS.

Memory Usage

ACE is about 570_8 bytes long.

Usage

ACE may be called through CPC by the CONTRLC macro or by placing ACE in RA+1 or an input register. ACE must be called with the auto recall bit set to prevent a central program from using the input file. In the lower 18 bits of the call is a pointer to the function/status word. This word should be zero except for the fifth byte which contains the function code.

Upon return the status will be set odd. Also bit 4 (20_8) will be set for end of record.

Residence of Routine

ACE should be disk resident to minimize CM requirements as the usage will be small.

APR Automatic Program Sequencer

The Automatic Program Sequencer allows the regular rerunning of jobs. Jobs are entered under the sequencer via control cards, and at the completion of execution are saved by the sequencer to be executed again at some future time. The control cards serve to place the jobs under the control of the sequencer, to supply the interval or execution frequency of the job, and to provide the sequencer with certain "housekeeping" necessities. Operator control of the sequencer is provided via console entries, and the sequencer itself is a peripheral processer program (APR) and a table in CMR (T.SEQ).

The CMR table T.SEQ is used by the sequencer program APR as a working storage area. It contains two types of entries. One for the APR program and one for each of the jobs running automatically under the sequencer. The first entry in the table is the APR entry. Each succeeding entry is for a job. The entries are numbered 00-NN where NN is the maximum (octal) number of jobs allowed under the sequencer (L.SEQ-1).

Associated with the sequencer are a number of system pointers. These are:

T.SEQ.	The address in CMR of the diagnostic sequencer table.
P.SEQ.	Word 4 of CMR. Contains T.SEQ/10B.
L.SEQ.	The length of the table.
C.SEQ.	The byte in P.SEQ. containing the pointer to T.SEQ. This is in byte 2.
C.SEQL.	The byte in P.SEQ. containing L.SEQ. This is in byte 3.

Calls to APR

All calls to APR are processed by LOADER.

APR (0,0) Places APR in the delayed request stack. This call should not be made from a control card or through DIS. The call is made by APR itself on receipt of an APR (0,1) call or an APR (0,0) call from MTR. If this call is made as a result of APR (0,1) being called, the sequencer merely puts itself into the delay stack to be recalled by MTR in one minute. If called as APR (0,0) from MTR, the clocks for each job in the sequencer table (T.SEQ) are decremented. If any of the clocks is zero then the sequencer unlocks the file for that job and allows it to be advanced to a Control Point via begin-job processing.

APR (0,1) Turns the sequencer on and places APR (0,0) in the delayed request stack; delay is one minute. APR checks to see if the sequencer is already on, in which case it issues a dayfile message to the operator informing him that the sequencer is already on. This avoids having more than one copy of APR in recall. APR always goes into recall at control point zero. If it has been determined that the sequencer is not already on then APR proceeds to put itself into recall at control point zero with a 59.75 second delay.

APR (0,2) Turns the sequencer off. This call causes all job sequencing to cease. APR clears the on/off switch in the control entry of T.SEQ (entry 0) and drops its PP without going into recall. None of the jobs' entries in the sequencer table is altered. If the sequencer is subsequently turned-on then the rerunning of jobs will continue as before.

APR (1,XXXXNN) This call is usually made via a control card in a job. It makes this job number NN in the sequencer table. Interval will be XXXX octal minutes. NN is checked in subroutine CKN to insure that NN is within the limits of T.SEQ. Next the drop flag is checked to see if the operator has elected to remove this job from the sequencer table. If the flag is set then the job is dropped and the job file released via end-job processing. If NN is legal and the drop flag is not set then a check is made to insure that the job was not entered before. If it has then XXXX is disregarded. If the job is not already in the sequencer table then the following occurs: S.CPS is set in W.CPERT in the control point area and 1EJ will later make the file a rewound, locked, input entry. The FNT address is saved in the sequencer table for this job and the interval and clock bytes are set to the value of XXXX. The number of sequencer jobs in the table is incremented in entry 0 and the PP is dropped. $XXXX \leq 3777B$ and $1 \leq NN < L.SEQ$.

APR (2,NN) Runs job NN now. NN is checked by CKN, and if valid, then the lock bit in the FNT entry for this job is cleared and the job is allowed to advance to a control point via begin-job processing. The clock byte in the sequencer table is not altered for this job and it will be run again at its schedule time. $1 \leq NN < L.SEQ$.

APR(3,NN) Drops job NN. NN is checked by CKN and if found to be valid APR will set the drop flag for the job NN in the sequencer table. The drop flag is checked when the next execution of the job takes place. The job is executed this final time and is allowed to drop via end-job processing. The entry for this job in T.SEQ is then removed.

APR(4,XXXXNN) Sets the interval for job NN to XXXX octal minutes. NN is checked by CKN as usual and if found to be valid APR proceeds to set the interval byte in T.SEQ to the value of XXXX. XXXX and NN are extracted from the input register by subroutine INS. The clock byte is also reset at this time and the job will not be run at the time scheduled by the former interval. $XXXX \leq 3777B$.

APR (5,XXXXXX) APR will read the real time clock and place its value in byte 4 of CM address RA+XXXXXX of the control point of the calling job. If RA+XXXXXX is found to be out of range then an error message will be issued to the dayfile and APR drops its PP. The job is aborted as a result of RA+XXXXXX being out of range. RA+XXXXXX is left unaltered. $XXXXXX \leq 377777B$.

APR (6,NN) Lists in the Dayfile the diagnostic programs associated with job NN. The diagnostic bits for job NN are decoded and yield the name of the particular diagnostic program or programs that this sequencer job is running. Diagnostic program names are contained in a table within the APR program. The message format is as follows:

DIAG. SEQ. JOB NN CONTAINS
FST
ALS
CT3 (etc.)

APR (7,XXXXNN) Sets the bits for the diagnostics associated with job NN. NN is checked by CKN. The diagnostic bits for job NN are set to the value of XXXX. This call is not necessary just for sequencing jobs, but is used just to be able to list jobs with an APR(6,NN) call. The bit values for XXXX are:

0001	0002	0004	0010	0020	0040
CT3	MY1	CM6	CU1	ALS	FST

APR (10,XXXXX) APR will place the CMRA, ECRA, and ECFL in RA+XXXXXX of CM at the control point of the requesting job. RA+XXXXXX is checked to insure that it is within the field length of the control point. Again if RA+XXXXXX is out of range an error message is issued to the dayfile and the job is aborted without changing the contents of RA+XXXXXX. XXXXXX ≤ 377777B.
Format of the CM word placed in RA+XXXXXX is as follows:

byte 0 = 0
byte 1 = 0
byte 2 = CMRA/100B
byte 3 = ECRA/1000B
byte 4 = ECFL/1000B

APR (11,NN) Suppress output, dayfile, and separators of job NN. APR sets kill flag (F.ERK) in C.CPEF in control point area and drops job at control point. JANUS senses the kill flag and inhibits any output from the job.

Subroutines

RDT Read entries from the sequencer table (T.SEQ).

This subroutine is entered with the desired entry number in the accumulator. The ordinal is saved and is added to the value of P.SEQ to determine the absolute address of the entry in CMR. RDT then returns with the entry in D.TWO - D.TW4.

CKN Checks the value of NN in APR calls. NN is obtained from D.PPIRB+4.
Since NN is always the sequencer job number it must be as follows:
$1 \leq NN < L.SEQ$
If the above condition holds then CKN returns with the value of NN in the accumulator.

REQ Requests FNT/FST channels.
REQ issues FNT and FST channel requests from MTR and returns when both channels have been assigned.

DRP Drop FNT/FST channels.

DRP issues requests to MTR to drop the FNT and FST channels.

INS Assembles interval XXXX and job number NN.

INS extracts XXXX and NN from the input register and returns with XXXX in D.PPIRB+3 and NN in D.PPIRB+4.

LCF Lock file.

If the FNT address of the input file is known LCF is entered with the address in the accumulator. If the FNT address is unknown then LCF will call subroutine SCH to find the FNT entry. S.CPS is set in W.CPERT in control point area and 1EJ will later make the job a rewound, locked, input file.

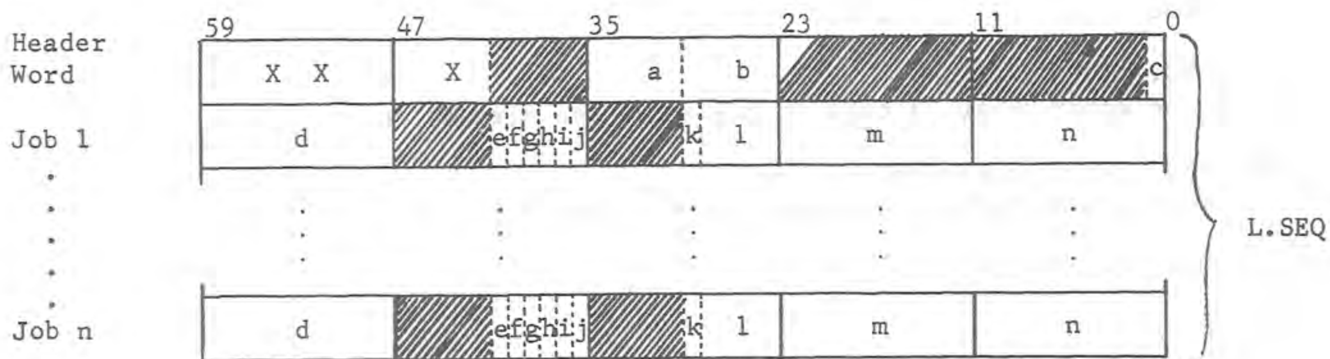
ULF Unlock file.

ULF is entered with FNT address in D.TW2. The entry is read in from the FNT table, the lock bit is cleared and the entry is rewritten. APR takes the precaution of calling REQ and DRP in performing the above.

SCH Search for *INPUT* file

SCH searches the FNT for an input file assigned to the jobs control point. When the FNT entry has been found SCH returns with the address in D.TW3.

SEQUENCER TABLE (T.SEQ)



XXX - APR in display code

a - (L.SEQ-1) number of jobs allowed in table

b - number of jobs in table

c - 1 = sequencer off
    0 = sequencer on

d - pointer to FNT entry for this job

e - FST flag
f - ALS flag
g - CU1 flag
h - CM6 flag
i - MY1 flag
j - CT3 flag

(e through j: these flags are set by the APR(7,XXXXNN) call)

k - 1 = drop this job after next execution; set by APR(3,NN) call

l - sequencer job ordinal: $01 \leq \ell \leq$ (L.SEQ-1)

m - job interval set by the APR(1,XXXXNN) call

n - clock that is decremented by APR(0,0) call

Messages Issued by APR

DIAG. SEQ. IS ON.
Sent after APR(0,1) call.

DIAG. SEQ. ALREADY ON.
Sent after APR(0,1) call if sequencer was already on.

DIAG. SEQ. HAS NO SUCH JOB.
Sent if APR(2,NN) requested and there is no job NN in the table.

DIAG. SEQ. JOB NN CONTAINS:
CU1
FST (etc.)
Sent by APR(6,NN).

T.SEQ LESS THAN 2 CM WORDS.
Attempt to execute APR with less than sufficient table length.

ILLEGAL ADDRESS REQUEST TO APR.
Sent when the address requested in an APR(5,XXXXXX) or an APR(10,XXXXXX) call is out of range for the control point. APR will abort the control point.

ILLEGAL DIAG. SEQ. PARAMETER.
Sent if the following conditions are not met:

For APR(P1,XXXXNN) $0 \leq P1 \leq 11B$
$0 \leq XXXX \leq 3777B$
$0 < NN \leq L.SEQ-1$

For APR(P1,XXXXXX) $0 \leq P1 \leq 11B$
$0 \leq XXXXXX \leq 377777B$

Keyboard Entries	Corresponding APR Call
SEQ,ON.	APR(0,1)
SEQ,OFF.	APR(0,2)
SEQ,RUN,NN.	APR(2,NN)
SEQ,DROP,NN.	APR(3,NN)
SEQ,LIST,NN.	APR(6,NN)
SEQ,INT,XXXX,NN.	APR(4,XXXXNN)

Where NN is an octal number greater than 0, less than L.SEQ, and XXXX is an octal number 0000 to 7777 inclusive. The SEQ, part of the entry, is processed through the normal DSD/IEE method, resulting in a jump in IEE to SEQX, which processes the remainder of the entry.

Processing of ON, OFF, RUN, DROP, LIST, and INT.

The first character of the entry is loaded from the keyboard buffer, and shifted left one and saved. Each successive character is shifted left one more, that is:

First character shifted left one
Second character shifted left two
Third character shifted left three
Fourth character shifted left four   etc.

until a separator is encountered. Each character after its shift is added to the preceeding character. This gives a sum for each entry:

ON = 126B	Ex.	Ø = 17B	LS 1 =	36B
OFF = 146B		N = 16B	LS 2 =	70B
RUN = 350B		Sum =		126B
DROP = 710B				
LIST = 1024B				
INT = 352				

This sum is then compared against a table to determine if the entry is one of the acceptable ones. If not, a format error will be declared. If the entry and separator are acceptable, a jump is made to process the numeric part of the entry.

For ON, OFF, create APR(0,Z)   (Z = 1 or 2)

For RUN, DROP, LIST, create APR(Z,NN)   (Z = 2, 3, or 6)

For INT create APR(4,XXXXNN)

The word created is then placed in the PP message buffer, and a monitor function 37B (request peripheral job) is executed.

There is no information on this page

CHK Check Output File

CHK may be called to determine whether or not a file named OUTPUT exists at this control point. The address of a status word is given in the low order 18 bits of the input register. CHK searches the File Name Table for a local file named OUTPUT attached to this control point. If such a file is not found, a flag will be set equal to one. If the file is located, its position is tested. If the file is rewound, a flag will be set to one - otherwise the flag will be set to two. After the flag has been set, the routine stores the flag value in the status word specified in the input register and exits.

CTS COMMON File Processing

General

CTS is a PP program always loaded at 1000-. It is available to any CP or PP program that needs its services.

Function

CTS will make a local file common, or a common file at this control point local, or detach a common file at this control point and leave it free at control point 0, or bring a common file at control point 0 to this control point. On the choice of action, see ≠entry information≠ below.

Entry Information

The input register has the form:

```
VFD 18/OHCTS,3/0,3/n,18/0,18/a
```

where n is the control point number, and a is the address relative to RA of the request in central memory. This request has the form:

```
VFD 42/name,18/t
```

where "name" is the name of the file to be handled, left justified with zero fill, and t is:

- 0 if there is a local file with that name, which is to be made common, or if there is no such local file, bit a free common file of that name is to be brought to this control point,
- 2 if there is a common file with that name at the control point, which is to be changed to a local file,
- 4 if there is a common file with that name at the control point, which is to be re-assigned to control point zero and so left free.

Any other value of t is treated as if t=4, bit if the value is odd, a requested re-call will be ineffective, and the calling program will not know when the action is completed, and will probably mis-interpret the response.

Exit Information

The value of t in bits 0-17 of the CM request word is altered as follows:

a. if t was initially 0, it becomes:

   11B if there was already a common file with the name at the control point.

   7 otherwise, if there was a local file with the name, but also any common file with the same name.

1 otherwise, if there was a local file with the name, which has been converted to common, still at the control point; or if there was no local file with the name, but there was a free common file with the name, which has been brought to the control point.

13B instead of 1, if there was no local file with the name, but there was a free common file, but its equipment was not available for assignment to the control point.

3 if there was no local file with the name, and no free common file with the name, but there was such a common file assigned to some other control point.

5 if there was no local file and no common file with the name.

b. If t was initially 2 or 4, it becomes 1 if there was a common file with the name at the control point, which has now been made local or free common. If there was no common file with the name at the control point, t becomes 3.

Other Programs Called

None.

Narrative

First we must find out what files with the given name are in the FNT. We fetch the request, then request the FNT pseudo-channel, as we may be going to alter an FNT entry, and then zero cells FLOC, FCOMF, FCOMNF, and FCOML. Then, in the routine between CTSA and CTSK-1, we scan the FNT, disregarding entries without the required file name. FLOC receives the address of the FNT entry for any local file at this control point; FCOMF for any common file at control point 0; FCOML for any common file at this control point; and FCOMNF for any common file at any other control point. At CTSK, if the request type is not 0, we branch to CTD. Now if FCOML = 0, there is nothing to be done; we set the response to 3 and go to EXIT to write back the response, and drop the FNT pseudo-channel and the PP. If FCOML is not 0, we make the proper change in the FNT entry,depending on whether t=2 or 4, set the response to 1, and go to EXIT.

If the request is type 0 and FCOML is not 0, we set the response to 11B because there is already a common file with the name at this control point, and go to EXIT.

If the request is type 0 and FCOML = 0, we go to CTSKA. Now if FLOC = 0, we go to CTSD; what is wanted is to bring a free common file with the given name to this control point.

Then if FCOMF = 0, we cannot do so.

We set the response to 3 if there is such a common file at another control point, or otherwise to 5, and go to EXIT.

If FCOMF is not 0, we change the control point number in the FNT entry it points to our control point number, but do not yet write it back to the FNT. Now if the equipment is allocatable, or if it is not, but we successfully request it, we write the FNT entry back, set the response to 1, and go to EXIT. Otherwise, we send a dayfile message, set the response to 13B, and go to EXIT.

If the request type 0, FCOML is 0, and FLOC is not zero, we want to convert the local file FLOC points to into a common file. First we make sure FCOMF = FCOMNF = 0, i.e., that there is not already a common file of the same name. If they are not both zero, set the response to 7 and go to EXIT. If they are both zero, change the file type in the FNT entry FLOC points to common , write it back to CM, set the response to 1, and go to EXIT.

Subroutines

LDCA - this merely does what the macro LDCA D.PPIRB+3 would do.

Messages

Before returning the unsuccessful response 13B, the dayfile message

XX UNAVAILABLE

is issued. XX is the primary device number of the equipment assigned to the common file, as shown in its FNT entry.

Index to location symbols in CTS flowcharts

Name	Page	Name	Page
CO1	3	CTSKA	2
CTS	1	CTSL	2
CTSA	1	CTSM	2
CTSB	1	CTD	2
CYSC	1	CTDA	2
CTSD	3	EXIT	2
CTSDB	3	EX3	2,3
CTSE	3		
CTSF	3		
CTSG	1		
CTSH	1		
CTSJ	1		
CTSK	2		

Then if FCOMP [illegible] we cannot do so.

We set the response to [illegible] if there is such a common file at another control point. If otherwise [illegible] and go to EXIT.

If FCOMP is [illegible] we change the control point [illegible]. This entry is pointed to our control [illegible] number [illegible] is back to the FNT. [illegible] assignment is all [illegible] done, but we subsequently [illegible] file, [illegible] FNT entry [illegible] the response [illegible], and go [illegible] otherwise [illegible] and [illegible] messages, set the response [illegible] EXIT.

If the request type is FCOMP [illegible] and FLCC [illegible] we want to convert the local file FLCC points to [illegible] a common file. First we make sure FCOMP [illegible] FCOMNT [illegible] that there is not already a common file of the same name. If they are not [illegible] exit with the response [illegible] and [illegible] to EXIT. If they [illegible] each [illegible], change the file to [illegible] in the entry [illegible] a common [illegible] back to CM, set the response to [illegible] and [illegible] EXIT.

Subroutines

[illegible] does [illegible]

Storage

Before returning the [illegible] in [illegible]

[illegible] EBP

[illegible] (FNT) [illegible] in the common file [illegible]

Notes to Location Symbols in [illegible]

Name	Tag	[illegible]
		[illegible]

Ne[illegible]

Exit Information

None

Function

The input register contains

18/OHDMP, 3/0, 3/n, 18/x, 18/y

where n is the control point number, x is the starting address of the area to be dumped, and y is the last address of the area to be dumped. Four types of dump can be distinguished; they are prefaced in the listing with DMPX, DMPC, DMPA, and DMP respectively.

DMPX. If x=y=0, the exchange package in the control point is dumped, with P, RA, FL, EM, and the registers appropriately labelled, and the contents of the words to which the 8 A-registers point also listed. RA+0 through RA+77B are then dumped, numbered relative to RA, and the address at which the CP program stopped is determined; let us call this J. If P in the exchange package is non zero, J=P. Then cells RA+J-100B (or RA+100B if J is less than 200B) through RA+J+77B are dumped.

DMPC. If x=y≠0, the 200B cells of the control point are dumped, numbered 0 through 177B.

DMPA. If x is greater than 377777B, it is reduced by 400000B. Then if y is greater than 377777B it is similarly reduced. Then cells x through y-1 are dumped, numbered absolutely. But if x is greater than y after reduction by 400000B as necessary, the dump is terminated.

DMP. If x is less than 400000B, cells RA+x through RA+y-1 are dumped, numbered relative to RA. But if x is greater than y, or if y is greater than the field length of the job, the dump is terminated.

DMP can be aborted either by an operator drop or an I/O error. A message will be issued - DMP ABORTED - prior to exit.

DMP has been given the capability to produce labelled and change dumps when requested by the user. The following is a general description of the process as it differs from the original DMP program.

1. The input register is read. If bits 0-35 contain all ones, then DMP was called by control card. The routine ARG is called to fetch the parameters starting at RA+2. ARG also checks bits 28-29 of RA+66 to determine whether a labelled or change dump is requested. If the input register does not contain all ones in bits 0-35, then DMP was called by MTR and no labelled dump will be given. The dump parameters, in this case, are taken from the input register.

DMP Routine to Dump CM

General

DMP is a PP program always loaded at 1000B. It dumps one or more sections of central memory, in octal representation, on the file OUTPUT belonging to the same control point, for eventual listing.

DMP is called in one of three ways:

1. By a control card with one of three forms:

   DMP. (equivalent to DMP (0.0))
   DMP (y) (equivalent to DMP (0.y))
   DMP (x.y)

2. By SCOPE for a job that is aborted, before any control cards that may follow an EXIT card are obeyed. The call is equivalent to DMP (0.0).

3. By a CP program that puts into its RA+1 a word of the form VFD 18/OHDMP, 6/0, 18/x, 18/y

No recall is possible; i.e. the CP program cannot know when the action has been completed. Bit 40 of the word must be 0, as in the above format. If it were 1, SCOPE would look at the word in RA+y, and if bit 0 of that word were 1 at the critical moment, it would be as if bit 40 of the request word had been 0 in the first place. But if bit 0 of the word at RA+y were 0, the CP program would be hung up.

Note also that words RA+70B through RA+74B are temporarily disturbed, as explained in the next section.

Other Programs Called

OPE is called into another PP toopen the OUTPUT file. If the OUTPUT file is on a non-allocatable device, then calls are made to CIO to write formatted data from central memory to the OUTPUT file. To get the central memory space needed for the buffer area and FET, DMP requests as much storage as it can have up to the CM equivalent of its PP buffer area. The minimum amount acceptable is 300 octal CM words. If the OUTPUT file is on an allocatable device, then DMP issued stack requests to write the formatted data from the PP buffer to the file.

The FET for this CIO call is put temporarily at locations RA+70B through RA+74B. They are restored after the FET shows the action has been completed.

Entry Information

Only the call in the PP input register.

2. If a labelled dump is requested, ARG reads the DEBUG file and builds a list of all program and COMMON names and their starting addresses.

3. The routine ELB is called to output the dump. If a labelled dump is requested, the list of names prepared by ARG is used to output the labels.

4. The routine CHD is called to output the change dump if requested. A change dump is always preceded by a labelled dump.

Core allocation of DMP is as follows. Addresses are approximate.

1000	(DMP)	Start of permanent code
4100	(DBSTT)	Start of output buffer and code used for initialization (ARG,FNT,OUTS)
6200	(IBUF)	Start of input buffer. The output buffer also uses this space if there is not to be a labelled dump.

Step

This suboutine is really STEPA, but macro "STEP" is defined as RJM STEPA. It puts the contents of the A-register, 2 characters to be added to the dump, into the cell in the buffer to which BIAB points, and then increases BIAB by 1. If BIAB now contains DBSTT+ buffer limit, the buffer is full and subroutine DBUF is called to write it out and reset BIAB to contain DBSTT.

Entry Information

Two characters in A, and a buffer pointer in BIAB.

Exit Information

BIAB updated.

Subroutine Called

DBUF

Registers Destroyed

Only those DBUF destroys if it is called.

STUFF

Converts bits 0-5 of the A-register, considered as two octal digits, to the two corresponding display code characters in bits 0-11 of the A-register.

Entry and Exit Information

Only the contents of A.

Register Destroyed

COUNT2

ENDLINE

When this is called, BIAB points to the next free cell in the PP buffer beginning at DBSTT. The subroutine adds from 1 to 5 pairs of blanks to the buffer, updating BIAB, until the buffer contains an integral multiple of 5 PP words, i.e. an integral number of CM words. To find out how many pairs of blanks to add, the number of bytes already in the buffer is divided by 5, by subtracting 50B and then 5 repeatedly. The quotient is not interesting. If the remainder is 0, add 5 bytes, otherwise add 5-remainder bytes.

Entry Information

Buffer pointer in BIAB.

Exit Information

BIAB updated.

Subroutine Called

STEP

Registers Destroyed

COUNT, and those that DBUF destroys if STEP calls DBUF.

PUTBLAN

This is called with a number in the A-register indicating how many blank bytes are to be added to the PP buffer. The subroutine merely adds the bytes.

Entry Information

Number in A-register, buffer pointer in BIAB.

Exit Information

BIAB updated

Subroutine Called

STEP

Registers Destroyed

COUNT2, and those that DBUF destroys if STEP calls DBUF.

ELB

This dumps central memory, from the word addressed by the sum of the contents of RAFETCH+1 and IA, IA+1, to the word next before that addressed by the sum of the contents of RAFETCH, RAFETCH+1, and TA, RA+1. IA, IA+1 are stepped progressively until equal to TA, TA+1. (Let us call these simply IA, TA and RAFETCH). Basically, the words are dumped four to a line, the first of a line being one whose address relative to the content of RAFETCH is divisible by 4; the address of this word relative to the content of RAFETCH is printed at the beginning of the line. When a word from CM is the same as the preceding one, it is passed over, and this continues until an unlike word or the last word to be dumped is reached. The first word to be output after skipping is put in the next available column, and is preceded by its address relative to the content of RAFETCH. If the next available column is the first, the address appears normal except that it may not be divisible by 4. In any other column, the address will be separated from the word by a right arrow instead of a blank. It is only when it comes after some skipped words that a word whose relative address is divisible by 4 can appear in the 2nd, 3rd or 4th column. Otherwise, any word with such an address is put in the first column, even if this leaves one or more columns vacant on the preceding line.

To begin ELB, first test is made to see whether TA is greater than IA, and immediately exit if not. This precaution would appear to be unnecessary, but in fact if the CP halts with P=1, and a DMPX is done, the logic between GOP and E05 in this program will produce TA=IA=100B before calling on ELB. Probably this is the only such possibility, but the safeguard is in ELB in case ther are others lurking.

At ELA the word IA points to is fetched. Just before ELM there is a test whether this is the first time the sequence beginning at ELA has been executed. (It ought to test whether the first time since soubroutine ELB was called, but it tests whether the first time since DMP was loaded. The only time ELB can be called twice for one loading of FMP is in the case of DMPX, and then the flaw in the logic may result in skipping over the first few words when dumping P-100B to P+100B.) If so, things proceed as if the word were different from the preceding word and ELB goes to ELM. If not, it goes to ELM1 and matches the word with what is stored at BCOM through BCOM+4 (this is the preceding word). If bits 24-59 match, and bits 48-59 are 6000B, it is assumed that the word is a kind of pre-fill with which unused memory is stuffed by some loaders. The low order 17 bits would be the address of the cell itself relative to RA, and so though the word is negligible, it is not identical with the preceding word. (As a floating point number, the word is negative indefinite. As instructions, it is SBO=0 and PS"). So now without actually being sure that bits 24-47 contain

00000000B, nor that bits 12-17 of the word are correct, bits 0-11 are matched against the contents of IA+1, the low order part of the relative address of the cell. If this matches, go to ELM2, otherwise to ELM. If bits 24-59 match those of the preceding word but bits 48-59 are not 6000B, something more comprehensible is done, match bits 0-23, and go to ELM2 for a complete match, and otherwise to ELM. Of course if bits 25-59 do not match, we go to ELM.

At ELM2 set ELSET=1 to show that at least one word is being skipped, so that the next time a word is output it will be preceded by its address and maybe an arrow. Then go to ELC as if the word has been dumped.

At ELM copy the new word into BCOM through BCOM+4 to replace its unlike predecessor, and serve for comparing the next succeeding word.

Now if ELSET contains zero, skipping has not just been taking place, so go to EL13 to find out whether to print the address of the word relative to the content of RAFETCH, i.e., whether it is going to fall into the first column. In cell WORDS there is a column count that was set=1 at the very beginning of DMP, and which always contains the number of the next available column, between 1 and 4. If it now contains 1, go to EL15 to print the address. If not, get to EL131 and check whether IA+1 is divisible by 4. If so, any free columns must be skipped and the current word put in column 1 of a new line, so call subroutine ENDLINE to terminate the current line, set WORDS=1, and go to EL15. (This will happen if cell 1000B contains something that is dumped in column 2; 1025B and 1026B get dumped in columns 3 and 4, and 1027B is dumped in column 1 of the next line. Now 1030B comes up, and if different from 1027B, it is dumped in column 1 of yet another line.)

If the word is definitely not going to be dumped in column 1, come to EL14. It is going to be preceded by four blanks, not by any sort of address, so call subroutine PUTBLAN to add these blanks and then go to EL18 to format the word itself.

Come to EL15 when the address before the word itself must be output, whether in column 1 or not. First format the address taken from IA and IA+1, and put it in the buffer. Then add either three blanks or one arrow to the buffer, depending on whether we are in column 1 or not. Then continue to EL18.

From EL18 to nearly ELC the word taken from CM is being formatted into four groups of five octal digits in display code, with a blank after each group, and the whole thing added to the buffer. Then the column count in WORDS is increased by 1. If it reached 5, call subroutine ENDLINE to end the line and reset WORDS to 1 to begin a new one. Come to ELC when a word has been completely dealt with, by dumping it and maybe its address, or by skipping it. Now add 1 to IA+08 and if not equal to TA return to ELA to work on the next word. If IA now=TA, exit immediately from ELB if ELSET=0, indicating that the last word looked at was not skipped. But otherwise, in order to dump the last word so that the user will not be confused about the terminal address of his dump, alter the copy of the word before the last in BCOM, back up the address in IA, and return to ELA. Now the last word will be re-processed as if it were different from its predecessor.

Entry Information

Initial and terminal address in IA, IA+1, TA, TA+1.
Base address in RAFETCH, RAFETCH+1.
Buffer pointer in BIAB.

Exit Information

Updated buffer pointer in BIAB. IA now=TA, IA+1=TA+1, TA, TA+1, RAFETCH, RAFETCH+1 unchanged. (Secondarily, DBUF by STEP).

After each time IA has been incremented, a check is made to see if it equals the next address in the name table. If so, the routine LEO is called to print the appropriate label message. Certain portions of the code in ELB have been placed in subroutines (WORDY and ADR), so that this code could easily be used by the change dump routine, CHD.

Registers Destroyed

IA, IA+1, D.BA through D.BA+4, also any that ENDLINE,PUTBLAN, STUF, STEP, DBUF destroy if called.

DBUF

DBUF writes out the PP buffer on the output file, using calls to CIO or R.WRITEP. DBUF is called each time the buffer is full, with 1000B CM words, using a function code O.WRP or WRITE in the request it puts out, and once while terminating DMP, to clear out the buffer and/or write the short sector, using O.WRPR or WRITE.

Entry Information

Buffer Pointer in BIAB. Function code at OPCODE.

Exit Information

None (BIAB reset to DBSTT).

Subroutines Called

R.WRITEP, REQPP, WAIT, TFLCOMPTE.

Registers Destroyed

D.TO through D.T4; also D.T5 through D.T7 by R.WRITEP and D.Z6 and D.Z7.

FCF

This is called by subroutine DAB, and by the main routine below DXR, to take the word in D.TO through D.T4 and add it to the buffer after formatting it into five groups of four display code octal digits, each preceded by a pair of blank characters.

Entry Information

The word in D.TO through D.T4; the buffer pointer at BIAB.

Exit Information

Only the updated buffer pointer, at BIAB.

Subroutines Called

PUTBLAN, STUF (also STEP by PUTBLAN and DBUF possibly by STEP)

Registers Destroyed

COUNT (also COUNT2 by PUTBLAN and STUF, and those that DBUF destroys if called by STEP).

DREG

This is called by the main routine below DMPX, while dumping the exchange package, four times - to format P, A0, B0; RA, A1, B1; FL, A2, B2; and EM, A3, B3, as well as the contents of the cell pointed to by each A-register. Before calling DREG, the word of the exchange package has been fetched to D.T0 through D.T4, the A-register part of the word has been saved at ASAV (except in the case of A0), and the P, RA, FL, or EM, has already been put into the buffer with subroutine STEP. DREG, on being called, now puts a pair of blanks in the buffer, and then formats the number from bits 36-53 of the exchange package word, and puts the six octal digits in the buffer after PP, RA, RA+FL, or EM. Then it adds another pair of blanks to the buffer. Then it calls subroutine DAB to format and write out the contents of the A and B registers and the word the A-register points to.

Entry Information

A word from the exchange package in D.T008 through D.T4; the buffer pointer in BIAB.

Exit Information

The buffer pointer in BIAB updated. D.T0 through D.T4 unchanged.

Subroutines Called

PUTBLAN, STUF, DBA (also ENDLINE and FCF by DAB; STEP by all of them. Note that DBUF is never called by STEP, as DREG is only called for a DMPX-type dump, and this kind of dump cannot fill the PP buffer).

Registers Destroyed

None by DREG directly; COUNT and COUNT2 by PUTBLAN, STUF, DAB, ENDLINE, FCF, STEP.

DAB

This is called by subroutine DREG to dump the A and B registers numbers 0 to 3, and the contents of the words of the A registers point to,and by the main routine below DEX4 to dump registers 4 to 7. If DAB is called at all during DMP, it will be called exactly eight times; and the A-register number at DA1, the B-register number at DBB1, and the A-register number at DCA1

for C(An)=IN the dump are all increased by 1 each time through, immediately after being used. The logic of DAB is completely straight forward. The word from the control point area is already in D.TO through D.T408 when DAB is called, and the A-address portion of it has already been saved in ASAV and ASAV+1. After the B-address portion has been formatted and put in the buffer, TA which contains the address of the exchange package word in CM, is compared with the address of the control point. If they are the same, the A-register was A0, which is not considered to point urgently to anything; so subroutine ENDLINE is called to round off the line in the buffer and then the program exits from DAB. Otherwise DAB continues to fetch the word that ASAV and ASAV+1 point to, format it, and add it to the buffer before calling ENDLINE and exiting.

Entry Information

A word from the exchange package in D.TO through D.T4; the A-register portion of this in ASAV and ASAV+1; the buffer pointer in BIAB; the CM - address of the exchange-package word in TA.

Exit Information

The buffer pointer in BIAB updated.

Subroutines Called

STEP, PUTBLAN, STUF, FCF, ENDLINE.

Registers Destroyed

COUNT, COUNT2, and D.TO through D.T4.

Note that DBUF is never called in connection with DAB, for the same reason as explained above for subroutine DREG.

OUTS

This subroutine is called only once, rather early in the main routine, to locate or create the FNT entry for the output file on which the dump is to be written.

ARG

This routine is always called when DMP is called by control card. It does the following:

a. Checks bits 28-29 of RA+66 and sets a flag (DUMPFLAG) = 1 if a labelled dump is requested and = 3 if both labelled and change dumps are requested.

b. If a labelled dump is requested, FNT is called to find the FNT/FST entry for the file DEBUG. The file is then rewound and read to the end of the first record (loader table record). As the file is being read, a name table consisting of 5-word entries is built in the following format:

IBUF +1 +2	+3 +4
NAME	Adr
NAME	Adr
. . . .	. . . . . .
NAME	Adr
0	Adr

If the DEBUG file has to be reread before reaching end-of-record, the origin of the input buffer is moved up because the table is built at the start of the input buffer. Word one of each 5-word entry will have one of the following meanings:

Zero: This is the last entry in the table. The address refers to the LWA of the load.

Bit 59=0: The name in bits 18-58 is the name of the program.

Bit 59=1: The name in bits 18-58 is the name of labelled common block.

7700_8: This entry refers to blank common.

c. The parameter count is fetched from RA+64. If a labelled dump is not requested, the maximum allowable number is two, but if a labelled dump is requested, there may be up to four parameters, because of the use of the empty parameter preceding a name to designate a common block name. The parameters are picked up, starting at RA+2, and converted to octal. ARG exits after a value has been stored at both IA (Starting dump address) and TA (ending dump address). Alpha-numeric parameters are allowed only if a labelled dump has been requested. The table of names is searched for the corresponding name and address. If the name is not found in the table, a dump ARG error results.

DMP4

This routine processes the DMP ARG ERROR message. It has been changed so that DMP does not abort, but merely ignores the dump, drops the PP, and exits to the idle loop.

MOVEIN

This small routine is used by ARG to place a program or labelled common name in the name table. In order to determine whether labelled common addresses have already been stored in the table, this routine maintains a location (LOC) which contains the value of the highest address stored in the table.

BLCM

This small routine is used by ARG to place the blank common indicator and address in the name table. After the store is performed, the blank common indicator (BL, BL1) is cleared to prevent more than one call to this routine.

LEO

This routine is called by ELB whenever it is time to print the next labelled dump message. Printing of the proper message is controlled by a flag (TT) which is changed by both ELB and LEO. The flag has the following meanings:

0 IA is still less than the first address in the name table, so the message "LOW CORE" is printed.

1 IA is between the first and last address in the name table, but it is necessary to use the previous table entry, since this is the first entry to LEO, and IA was initially between two program origins. The program name (or // for blank common) followed by the message "PROGRAM" or "COMMON" will be printed.

2 Same as above, except it is not necessary to use the previous table entry.

3 IA is equal to or greater than the last name table address. The message "END OF LOAD" is printed.

ADR

This routine is called by ELB and CHD to place the absolute address in the print line. This function was previously imbedded in ELB.

WORDY

This routine is called by ELB and CHD to format a CM word and place it in the print line. This function was previously imbedded in ELB.

The format of the print line had to be changed to make room for the relative address counter. The rightmost character now goes in print position 136, where it previously went in position 129. The relative location counter (if the dump is labelled) goes on the extreme left side of the line.

CHD

This routine prints the change dump, if one is requested. The following takes place:

a. IA is reset to the requested FWA of the dump.

b. The DEBUG file is positioned to the end of the first record and is read. (The name table used for the labelled dump is now lost.) A pointer to the word in the input buffer (5 PP words) is established which corresponds to IA.

c. The CM word at IA is fetched and compared with the corresponding word in the input buffer.

d. If the words are equal, IA and the buffer pointer are incremented, and step c is repeated.

e. If the words do not compare, ADR is called to print the absolute address IA, and WORDY is called twice, first to print the current value of the CM word at IA, and then the value as indicated in the input buffer.

f. Steps c-e are repeated until IA equals TA.

FNT

This routine is called by ARG to find the FNT address of the file DEBUG. When found, the FNT address is stored in the proper field of the request that will be used by READ to read the file.

If an entry for the file cannot be found, or if the entry designates that the file is on a non-allocatable type device, the special dump flag is cleared. This will result in the dump being a regular unlabelled dump.

REWIND

This routine rewinds the file DEBUG by setting the appropriate fields in the FST. The current RBA is reset to the first RBA, and the current RBT and current PRU fields are set to zero.

READ

This routine issues a call to R.READP in order to read the file DEBUG to PP memory. On entry to this routine, the accumulator contains the desired FWA for the read. This will normally be IBUF, but will be a higher address on successive reads to a long loader table record, while ARG is building the name table.

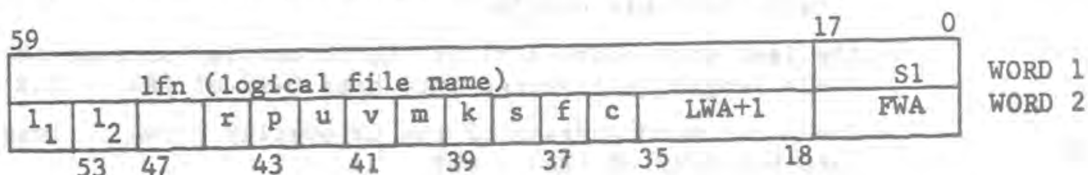
LDV ABSOLUTE OVERLAY LOADER

This PP overlay is called as a result of the user's program issuing the following request:

LOADREQ, param

I. OVERLAY LOADING

If the value of "param" is non-zero, it points to a two-word area in the user's field length containing the following information:

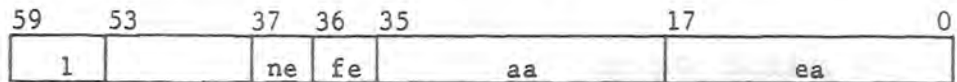


The full significance of this two-word area is described in Chapter 3 of the SCOPE 3 Reference Manual. Of significance here are the fields lfn, lwa+1, fwa, v, and u as all other fields are ignored. For "param" ≠ 0, the operation proceeds as follows:

1. If v = 0, this is not a request to load overlays. This unauthorized use of LOADREQ will result in a call to LDR.
2. If v = 1 and u = 0, this is a request to load an overlay from a user file and results in a call to the PP overlay LDR. LDV will replace the name LDV in bits 42-59 of its input register with the name LDR. The input register is restored to central memory and and exit is made from LDV with a jump to R.IDLE.
3. If v = 1 and u = 1, this is a request to load a CP overlay from the system library. This procedure is as follows:
   a. A request is made by LDV to drop the CP.
   b. A backward search is made of the portion of the Program Name Table (PNT) in the system library directory containing names of CP overlays. The CP overlay sought after by this request is named lfn. This search is executed with a return jump to a closed subroutine within LDV, DIRSCH.
   c. If upon return from DIRSCH, (A) = 0, no entry had been found in the PNT by this name, an error diagnostic is issued to the dayfile and the user's job is aborted.
   d. If upon return from DIRSCH, (A) ≠ 0, (A) = the CP location in the PNT containing an entry for this name. This entry had been read by DIRSCH from central memory into PP memory at locations D.T5 through D.T5+9.
   e. A space check is made: Is overlay size (defined by the 18 bit field contained in the lower half of D.T5+3 and in D.T5+4) less than or equal to lwa + 1 - fwa? If not, a diagnostic is issued to the dayfile and the user's job is aborted.
   f. The residency check is made: If CM resident (defined by bits 7-4 of D.T5+5 being 0), the overlay is moved from CMR into user's area beginning at (RA) + fwa. If disk resident, a stack request is formatted and entered into the stack for the mass storage device.

g. Upon completion of loading the overlay into the user's area, the reply information is generated.

-WORD1 is cleared to zero
-WORD2 is filled with the following information



where l, ne, fe, and aa are set to zero and ea = entry point address is taken from bits 0-17 of the 1st word of the 50 table for this overlay.

- the last word address +1 of the CP overlay storage in user's field length is inserted into bits 0-17 of (RA) + 65B
- The first word address of the CP overlay = fwa is inserted into bit 0-17 of (RA) + 66B
- all other previous information in (RA) + 64B through (RA) + 67B is retained.
- a request is made to recall the CP
- a request is made to drop the PP
- an exit jump is made to R.IDLE

II. SPECIAL CALL FOR COMPILE LOAD AND GO OPERATION

A RUN(G) control card causes a LOADREQ request to be issued with param 0. The occurrance of this option will result in a special use of the LOADREQ request. LDV will do the following:

1. Insert a 64B into bits 0-17 of the PP input register.

2. Obtain a byte from a word in the control point area. This word is referenced by the label W.CPLDR. The byte begins in bit 19 of this word. The content of this byte is an indication as to which loader is currently used by the job executing at this control point.

3. This value in this byte will be used as an index to be added to the first word address of a table of pointers.

   (byte) + TBLPTRS → A

The resulting address points to the 1st of two PP locations containing the 3-character name of a PP overlay. This name is inserted into bits 42-59 of the input register for this PP.

$((A)) \rightarrow$ P.P.I.R. bits 59-48

$_{11}((A)+1)_6 \rightarrow$ P.P.I.R. bits 47-42

4. This name represents the PP overlay which is called upon to load the loader selected for the job currently executing at this control point.

The PP input register is restored to central memory, and an exit jump is made from LDV to R.IDLE.

LOC Load Octal Cards

LOC will process the next logical record on the file named INPUT. A block of central memory may be optionally cleared prior to loading corrections. If the clearing option is to be used, the range is specified in the input register as follows:

Bits 18-35.. First word address of area to clear
Bits 0-17.. Last word address +1 of area to clear

Storage clearing is performed in the subroutine CLS. The FNT is then searched to locate the file named INPUT at this control point (FNTSRCH). Once the file is located, PP memory is prepared for a disc read into the PP (BUFINIT) and up to 5000B PP words are read (READPP). The routine, UNPACK, is called to extract one card image, i.e. one octal correction. AAD and ASW are then called to extract the address and word content, respectively, from the card image. The word is written to central memory and LOC loops back to UNPACK for the next card.

This process continues (with calls to READPP to read more data, if necessary until an end-of-record status is encountered.

MDI Move System Directory (EDITLIB)

General

MDI is a PP program, always loaded at 1000B. It is called only by EDITLIB, in connection with directory manipulation.

Function, Entry and Exit Information

The input register always has the form:

```
VFD 18/0HMDI,3/0,3/n,18/0,18/a
```

where n is the control point number and a is the address relative to RA of the first of two consecutive cells in central memory containing the request. These two cells are formatted:

```
VFD 12/t,48/0
VFD 30/c,30/d
```

Bit zero of the first of these words is set to 1 by MDI when it has completed its action. The nature of the action depends on t.

If t is greater than 4, the job is aborted.

If t=0, c in the request is ignored. The CM-resident directory is copied into cells RA+d through RA+d+k-1, and the two cells at RA+a and RA+a+1 are left:

```
VFD30/b,30/1
VFD30/k,30/d
```

K is the length of the directory, and b is the absolute address at which it begins in the CM resident. No check is made on whether k+k is greater than the field length of the job.

If t=2, the action is the same as for t=0, except that only the file name table in the directory is copied, instead of the whole directory. Then k is the length of this table.

If t=3, nothing is copied to or from the directory, but the two cells at RA+a and RA+a+1 are left as for t=0. This option is used by EDITLIB when it needs to know only the address of the directory. D and c in the request, and k in the response, have no significance.

If t=1, the CM-resident directory is to be replaced by the words in RA+d through RA+d+c-1. But first the operator is to be given a chance to abort the job. So MDI alters t in the request, in central memory, to 4; sets bit 12 of RA+0 to 1 (a type-in of n.GO can zero it), puts the message "EDITLIB WARNING - GO OR DROP" on the B-display, and recalls itself with one second delay.

If t=4, and bit 12 of RA+0 has been reset to 0, the replacement is carried out and the response in cells RA+a and RA+a+1 is:

```
VFD 30/b,30/1
VFD 30/c.30/d
```

where b has no significance.

Other Programs Called

1MD (see subroutine GRPQ)

PP-Resident Subroutines Called

R.MTR for storage request for control point 0, if the length of the CM-resident directory is changed.

Narrative

Note that there are macros defined for loading, subtracting, and storing CM addresses taken from 2 consecutive bytes. The 2-byte address holders are named by their first byte, and this is not specially noted below.

First we check the error flag, and drop the PP if it is set. Then check bit 12 of RA+0; if it is 1, we are waiting for a type-in of n.GO or n.DROP, and so branch to RECALL to recall MDI after one second. Otherwise, the request is brought from RA+a and RA+a+1 to TYPE through TYPE+9, and CM cell P.LIB is read into D.T0 through D.T4 and HOLD through HOLD+4. This means that the 2-byte address of the beginning of the directory is at D.T0 and HOLD, and its last+1 address is at D.T2 and for type 0, GP for type 1, GE for type 2, and GAB for HOLD+2. Then we branch on the type of request; to GA for type 0, GPRE for type 1, GE for type 2, GAB for type 3, or GP for type 4.

We go to GAB when all necessary copying has been done, or if type 3, immediately because there is no copying. The response will be copied back from TYPE through TYPE+9 to RA+a and RA>+A+1, but RA+a will be copied last to prevent the CP program from seeing the completion bit prematurely. First the completion bit is set to 1 in TYPE+4. Then the 2-byte address of the start of the CM -resident directory is moved from HOLD to bits 30-47 of the word in TYPE through TYPE+4. Then the 2-byte address in TYPE+5 and +6, which will be the length of what was copied if anything was, is moved to bits 30-47 of the word in TYPE+ through TYPE+9; the request is returned to central memory, and the PP is dropped.

Now for what is done before going to GAB if the request type is not 3. If the type is 0, we go to GA. Subtract the 2-byte address of the start of the directory from that of its end+1 and store the length in TYPE+5 and TYPE+6, from which it will be copied later into the response, and

also at BS and BS+1, where it will be counted down as the directory is copied. Then move the 2-byte destination address from TYPE+8, adding RA to it, and store at D.T2. Now copy the directory, in the loop from GAD to GAB-1. If BS contains 0, the move is completed and we go to GAB.

If BS contains not less than TH, which was preset to 10000B, go to GAC, reduce (BS) by (TH), and copy (TH) words from the address D.T0 points to the address D.T2 points to, and add(TH) to those address holders. Then return to GAD. But if BS contains less than TH, store (BS) in TH and return to GAD to try again. Then if BS contained zero, we are finished and go to GAB. Otherwise we come through the loop again, setting BS to zero this time, and move (TH) words, some number less than 1000B, and go back to GAD, whereupon we find (BS)=0 and finish.

If the type is 2 we have a similar action. But instead of using the pointers at D.T0 and D.T2 that frame the whole directory, we must first replace them with pointers to the beginning and end+1 of the program name table. So we go to GE and read the word D.T0 points to. This is the first word of the directory. Bits 0-17 of that word point to the word of the directory that sits between the end of the entry point table and the beginning of the program name table. We read that word into HOLD through HOLD+4, and store address plus 1 at D.T0, as the beginning address of what we have to copy. Now bits 0-17 of the word in HOLD through HOLD+4 are the address of the word that follows the program name table, so we move that address to D.T2 as the last+1 address of what we have to copy. Now we are nearly ready to go to GA and copy as if the type were 0. But HOLD 8 through HOLD+4 no longer contain a copy of D.T0 through D.T4, which is what we shall want after GAB, so we first make them contain such a copy, and then go on to GA.

If the type is 1 we go to GPRE, to change the type to 4, put the go-or-drop message in the B-display, set bit 12 of RA+0 to 1, and recall MDI with one second delay. If the byte is 4, and we have got this far, the operator has consented and we go to GP. First we get the length of the new directory and store it at TYPE+5 and TYPE+6, then add it to the address of the beginning of the CM-resident directory, and store what will be the new end+1 address in HOLD+2. Then we see whether this is at, above, or below the present field length of contol point zero. If it is at, there is no problem and we go to GPE, call subroutine GPW to cpy the new directory to CM resident, and then go to GAB to terminate. If it is below the present field length, we go to GPF; call subroutine GPW to copy the new directory; then call subroutine GPRQ to reduce the field length of control point zero; and finally go to GAB. If the new directory would extend beyond the present field length of control point zero, we must first call subroutine GPRQ to request a larger field length; then go to RECALL to recall MDI after 1 second delay. This will happen repeatedly until either increased field length is granted, or the operator drops the job.

Subroutines

The only subroutines are GPW and GPRQ, which are only called for type 1 requests.

GPW

This subroutine copies a new directory, whose address relative to RA is in TYPE+8 and whose length is at TYPE+5, into CM resident, beginning at the address in HOLD (and also in D.TO, but we are about to destroy that). First we zero CM cell P.LIB, and wait one second. The directory is already unavailable for CM loading, because file SYSTEM has been assigned to our control point. But we must make is unavailable to PP-residents before disturbing it. If a PP-resident finds zero in cell P.LIB, it waits for non-zero, which we shall not reset until the new directory has been completely copied. If a PP-resident finds non-zero just before we zero the cell, we give it one second to complete using the directory. This is safe because the PP-resident needs only to copy a CM-resident program out of the directory, which it can certainly do in less than a second unless there is some incredible jamming of the channels (pyramids) between central memory and PP-memory, or else to get the disk address of a program from the directory. No program on disk is ever destroyed by EDITLIB or MDI. Even if it is superseded, the older copy is still there and can be successfully read once the disk address has been obtained, which can certainly be done within the 1-second delay.

After the delay, we move the length of the new directory from TYPE+5 to BS, its starting address, with RA added on, to D.T2, and the starting destination address in CM. Resident from HOLD to D.TO. Then copy the directory with the loop at GPD and GPC just as we did above, in the reverse direction, at GAD and GAC. When this is complete, go to GPWX and copy the new directory pointers into cell P.LIB before exiting from the subroutine.

GPRQ

This subroutine calls 1MD to another PP to request a new field length for control point 0. The format of the call is:

```
VFD 18/0H1MD,30/0,12/a
```

where 100B times a is the new field length wanted.

Index to location symbols in MDI flowcharts

NAME	PAGE
GA	1
GAB	2
GAC	2
GAD	2
GP	3
GPC	4
GPD	4
GPE	3
GPF	3
GPRQ	4 (subroutine)
GPW	4 (subroutine)
MDI	1

1MD

1MD is a PP program, always loaded at 1000B. It is called only by MDI and only to control point 0, to request a change in the field length of that control point, in order to accommodate a changed directory.

The input register contains:

```
VFD 18/0H1MD,30/0,12/a
```

We request 100B times a words of field length for our control point (zero) and then drop the PP. Whether the new field length was obtained, if it represents an increase, will be checked by MDI.

MEM Process MEMORY Function

MEM - process a field length request from a central program. The request may be for a modification of the field length or simply a request for the value of the field length currently assigned. The request might also be for either the central memory field length or the extended card storage field length.

The input register will contain in bits 17-0 a relative address pointer to the parameter word and in bit 18 an indicator if the request is for CM or ECS. Bit 18 will have the value 0 if the request is for CM or 1 if it is for ECS.

Bits 59-30 of the parameter word contain the new value of the field length that is being requested. Bit zero is the status bit. It is set to the value one when the request has been completed.

MEM uses bit 18 of the input register to set its own internal type indication and to set an index at GETFL which is used to access of subroutine FCMFL to fetch the CM field length or FECSFL to fetch the ECS field length. It then fetches the current value and stores it af FLOLD.

The requested field length is read from the parameter word and if it is not already an even multiple of 100 CM or 1000 ECS words it is increased to the next higher even multiple. If the result is zero, the current value of the FL is loaded into the parameter word and the status is set to completed.

When a new FL is requested, MEM sets up the request to MTR and then tests if the request was granted. If MTR does not grant the request, MEM drops into the PP delay stack for a quarter of a second then comes back and tries again until it is successful or it finds the error flag set at the control point.

If the error flag is set at the control point it posts an error code in the status word and releases the PP.

REQ REQUEST CARD/FUNCTION PROCESSING

Philosophy

This version of REQ was rewritten for SCOPE 3.1.6. It is based on the earlier versions, but was rewritten to (1) clear up the code, (2) include new features, and (3) help reduce PP saturation by ending the need for repeated calling of REQ.

General Description

REQ is called either by 1AJ or by a central macro call to create a file and assign equipment to it. REQ has the following functions:

1. Ask operator for non-allocatable devices.
2. Assign non-allocatable devices by EST ordinal.
3. Ask operator for allocatable device, and assign it by device type or EST ordinal.
4. Assign allocatable equipment automatically by type or EST ordinal.
5. Set in FNT the request disposition codes.

A. REQUEST Cards

If called by 1AJ, REQ must decode the request. The parameters are left in RA+2, RA+3..by 1AJ with the number of parameters in RA+64_8. Each parameter is read and, based on its length, turned over to a section of code to break.*

B. REQUEST Function

If an external call is made to REQ (i.e., by a central program), the parameter list is read from central memory. Checks are made to insure the parameter address is in the user's fl, the status word is zero, and the auto recall bit is set.

---

* Except for the file name which must be the first parameter.

C. FNT Creation

At this point the information from either the card or function is in the following direct cells:

D.BA through D.BA+4	Logical file name, left justified zero fill.
D.FIRST+1	EST ordinal if specified
D.FIRST+2	8* For X tape 4 For 2MT,2HY,2LO,2HY 2* For Labeled tape
D.FIRST+3	Disposition code
D.FIRST+4	Device code and labels
ALLOT	Allocation style of mass storage

For both cases, the file name is verified for seven or fewer alpha-numeric characters, the first alphabetic. A search is made of the FNT. If no file at this control point has that name, an FNT entry is created . If one exists, the request processing is ended.

The disposition code is now entered in the FST. If it is MF for multi-file, the file name must be six or fewer characters.

If the request is for a multi-reel file, the secondary equipment byte in the FST is set to all seven's.

If the operator has pre-assigned equipment, an EST was specified in the request or automatic assignment of mass storage was specified, the flow proceeds as in section D. Otherwise, the request is displayed, the FST containing the above information is written, a pointer to the FST is left in byte C.CPREQ of the control point area as a flag to DSD and a check is made of the EST for the existance and availability of the requested equipment.

D. Equipment Assignment

When REQ is recalled by DSD {see Interface section}, the equipment assignment and pointer to the FST is read along with the FST entry.

---

* The data from a function call is mapped into D.FIRST+4 if a tape with only density specified.

If the operator has assigned an EST ordinal or an EST ordinal was specified, and the equipment is non-allocatable, the EST entry is read and the type cross matched. If the device is allocatable, the RBR's are searched for one of the proper type and allocation pointing to the assigned EST.

If the operator assigned mass storage by device type or automatic assignment was requested, the RBR's are searched for one of the proper type and allocation style. In this case, no EST ordinal is passed to 4ES through the FST.

If a second tape is needed, the FST entry is written along with the flag to DSD as before. Otherwise, the completed FST is written.

E. Termination

1. Normal - If the request is satisfied without errors, normal termination procedure is followed. If this was a control card request, REQ drops out allowing monitor to call 1AJ. If the request was external, the status reply word is set add and REQ drops out.

2. Abnormal - If error conditions occured and the request was by control card, the job is aborted. If the request was external, most errors will cause a status to be returned to the calling program.

   Normally, repreated calls to REQ will not occur in processing one request. However, if the FNT is full, or the request asks for a specific EST which is unavailable, REQ will bounce with a one second delay.

Major Subroutines (or sections of code)

DISPLAY	Writes the REQUEST message to the operator. for an external call, this routine must create the message. If a card call, the card image is used.
DISPDT	This routine returns in the A register a device neumonic for the octal device code in the A register when the routine is called.
REQ1AJ	Decodes the parameters from the request card.
FNT	Searches the FNT for the file name in D.BA. If it does not find one, it will create a local file of that name with an all zero FST.
EQAV	Searches the EST for the requested equip-

	ment type in DFST. IF available or assigned to another control point, EQAV will return. If all equipment of the request type is assigned to this CP, turned off, or does not exist, the request is ended with an abort or status code to the CM request.
SERIAL	This routine handles assignment of files to non-allocatable devices. Issues the assigned message and checks for multi-reel request.
COMPLT	Returns the status code to CM.
VFN	Verify file names and disallows "OUTPUT"
ALLOC	Assignment of allocatable equipment and equipment to files where no specific device was requested.
RBR	This routine searches RBR's for one with the proper device type, allocation style and EST ordinal.
FNTDC	This routine places the disposition code in the FST and if it is MF, checks the length of the file name and label bits.

Residence

REQ should, most likely, be disk residence. Since under normal conditions it will be loaded only twice per job the problem of disk saturation will not be increased.

Interfaces

The new concept of pseudo-activity in MTR is used to reduce bouncing of REQ. When REQ comes to a joint where operator action is required for equipment assignment, the pseudo-activity count is increased by one. This fuctions to prohibit monitor from calling 1AJ to the idle control point when REQ drops out. When REQ comes in again, this count is decreased by one.

When the operator assigns equipment REQ must be called in. All information needed to finish processing the request is stored in the FST entry. Thus, the address of this FST entry is placed in the control point area {byte C.CPREQ} before REQ drops out. This byte also serves as a flag to DSD to recall REQ. Then, when recalled, REQ can read up the equipment and pointer to the FST.

To be able to discriminate between type of calls and if REQ is being recalled, the input register format is changed. When an external call is made, the low order 18 bits of the input register contains the parameter address. When called by 1AJ, bit 41 of the IR is set. If the call is by DSD in response to operator assignment, byte three of the IR is non-zero.

RFL Request Field Length

RFL is called to modify the amount of storage assigned to a job. The request is in the following format.

RFL,fl where fl is the requested field length

The PP Input Register contains the following:

Bits 0-17	New field length requested
Bits 18-35	Address of the status response location
Bits 36-38	Control pointer number
Bits 39-41	Not used
Bits 42-59	RFL

The purpose of this routine is to request a new field length. The new field length is obtained from the Input Register and a request for storage is made. If the request for storage is satisfied and the status is zero, the PP is released, the dayfile message is cleared, and the routine exits to IDLE. If the storage request is satisfied and the status is non-zero, the status is written in central memory, the PP is released, the dayfile message is cleared, and the routine exits to IDLE.

If the request for storage is not satisfied and the error flag is set, the routine continues as if the request were satisfied. If the error flag is not set, the message WAITING FOR STORAGE is written, and RFL is recalled with a 250 msec. delay.

RFL Request Field Length

RFL is called to modify the amount of storage assigned to a job. The request is in the following format.

RFL,fl   where fl is the requested field length

The PP Input Register contains the following:

Bits 0-17   New field length requested
Bits 18-35   Address of the status response location
Bits 36-38   Control pointer number
Bits 39-41   Not used
Bits 42-59   RFL

The purpose of this routine is to request a new field length. The new field length is obtained from the Input Register and a request for storage is made. If the request for storage is satisfied and the status is zero, the PP is released, the dayfile message is cleared, and the routine exits to IDLE. If the storage request is satisfied and the status is non-zero, the status is written in central memory, the PP is released, the dayfile message is cleared, and the routine exits to IDLE.

If the request for storage is not satisfied and the error flag is set, the routine continues as if the request were satisfied. If the error flag is not set, the message WAITING FOR STORAGE is written, and RFL is recalled with a 250 msec. delay.

SRB Enter Expanded Disk Address into Directory (EDITLIB)

General

SRB is a PP program, always loaded at 1000B. It is available to any PP or CM program that needs its services, but only EDITLIB calls it at present.

Function

Given the name of a common file, and the information that would be in bits 0-23 of the second word of its FNT entry if it were positioned at the beginning of any record, SRB will furnish the RBT number and physical unit number for the record. Together with the 24 bits mentioned above, but without the name of the file, the information SRB furnishes suffices to address the record in a direct way.

When a record is written on disk for the first time by EDITLIB, the disc - driving programs return the 24 bits that were in bits 0-23 of the second word of the file's FNT entry just before writing began. For the convenience of PP-resident, if it will have to read the record from disk, the more complete address, independent of the file name, is wanted in the directory, and this is what SRB provides.

Entry Information

The input register contains:

```
VFD 18/0HSRB,3/0,3/n,18/0,18/a
```

where n is the control point number, and a is the address of the first of two consecutive words containing the request, relative to RA. The words RA+a and RA+a+1 are formatted:

```
VFD42/name,18/0
VFD 15/x,21/y,9/b,3/c,12/d
```

where "name" is the name of a common file, left justified with zero fill; x is ignored and will be left unchanged; y is ignored but will be replaced by the information whose furnishing is the whole purpose of SRB; b is the RBT ordinal of the record (left unchanged by SRB); c is the RBT byte number of the record (left unchanged by SRB); and d is the PRU number of the record (left unchange by SRB).

Exit Information

If a common file with the given name is not found, the request cells become:

```
VFD 42/name,18/1
VFD 60/0
```

otherwise, they become:

```
VFD 42/name,18/1
VFD 15/x,3/0,6/u,12/e,9/b,3/c,12/d
```

where x, b, c, and d are unchanged; u is the physical unit number for the record, and e is the RBT number.

Other Programs Called

None

Narrative

Between SRB1 and EXIT-1 we scan the FNT for a common file with the given name. If none is found, set the completion bit in the first request word to 1, zero the second request word, write them back to central memory, and drop the PP. If we find the wanted file, we arrive at SRB3, extract the RBT ordinal from the request and save it at D.FIRST, and then fetch the second word of the FNT entry into D.FNT-1 through D.FNT+3, so that the first RBT number is in D.FNT. Read CM word P.RBR into D.T1 through D.T5, so that the size of memory, needed for locating RBT word pairs, is in D.T5, and the base address for RBR's is in D.T1 and D.T2. The RBT word pairs for the file form a chain, the first byte of each giving the almost-address of its successor. Considering the links of the chain numbered, with the first one as number 0, we have to read link number b, where b is the RBT ordinal we have stored in D.FIRST. Link number 0 is addressed by the FNT entry, and we have read the relevant pointer into D.FNT initially. If we read the first word of each "link" (RBT word pair) into D.FNT through D.FNT+4, its onward pointer will be in D.FNT.

So we perform the loop of 11 instructions beginning at SRB3A b+1 times; fetching the first word of the next link in the RBT chain and counting down on D.FIRST until it is negative. At SRB3A each time through the loop, we have saved the pointer to an RBT word pair in D.EST+2 just before fetching the RBT word pair. So when we have the wanted pair, we have its almost-address in D.EST+2, which will be bits 24-35 of the second word of the response, or "e" in the format given for exit information. But it remains to find the physical unit number.

First read the second half of the RBT word pair. Probably the RBR number is in D.FNT+1, byte y of the word pair when its bytes are numbered X, Y, 0, 1 . . . 7. We put this in D.Z4 (at SRB51), and then scan bytes k through c inclusive of the word pair, where the bytes are imagined as numbered X, Y, 0...; k is the initial byte number given in byte Y of the RBT word pair, and c is the byte number given in the request. Each time we find a byte to be even, we know it is a new RBR link, and replace with it the previous value in D.Z4. On completing the scan, we know we have the RBR link for byte c in D.Z4.

Now, at SRB6, multiply this RBR link by 38, add it to the RBR initial address in D.T1 and D.T2, and read the RBR header word this points to. The physical unit number is taken from bits 42-47 of this word, and inserted in bits 36-41 of the second word of the request. Then we go to exit to set the completion bit in the first word of the request to 1, return the request to central memory, and drop the PP.

XDQ

Initial Call from DSD

When DMPQ.I (I for input, O for output or P for punch) is entered into the console, DSD forms a call to XDQ and places the parameter (I,O or P) in bits 0-5 of the PP input register XDQ checks the parameter and if it is less than 21B (display code P) + 1), XDQ assumes that DSD has called it and not the CM portion of the dump program.

XDQ checks if the parameter is I, O, or P and if not, the PP is dropped. The job name DMPQn (where n is I, O or P) is set into the control point area, and 5000B words of storage is requested.

The parameters for LOD are set into the field length and LOD is called to load XXXDMPQ. The file type parameter is set into RA+2 to be interrogated by the CM program.

The above description is also valid to loading XXXRESQ from XRQ.

Called from XXXDMPQ

XDQ is called from XXXDMPQ with the address of a three word buffer in bits 0-17 of the call. As the address will be greater than 21B, XDQ assumes that this is not the initial DSD call. XDQ interrogates bits 24 and 25 of its input register to determine what type of file is being dumped.

00	input
01	output
10	punch

Depending on the file type, various instruction modification is done.

Major Subroutines

1. SCFNT - Search FNT

   This routine searches the FNT for the specified type file. If one is found, the address of the entry is saved in D.FRO, if no entry was found, a zero is stored in D.FRO.

2. RCH - Request FNT Channel

   The internal error flag is checked, and if on, the file is closed and the PP dropped. If not, the FNT channel

is requested and the FNT entry whose address is in D.FRO is read into the PP. If it is no longer there, a call is made to SCFNT to get another entry. If it is still a valid entry, depending on the initial instruction modification, a check is made for disposition code of punch if the type is "OUTPUT". If it is type OUTPUT, a check is made to determine if the file has been back-spaced, if yes, the number of RBT bytes are counted and the total is placed in the field normally occupied by the address of the current RBT word pair.

The FNT/FST entry is then written to CM into the buffer specified by the call to SDQ. The file is then set to type local, the disposition code cleared and the file set to beginning of information. The FNT/FST entry is then written back to the FNT table. A dayfile message containing the job name is written to the dayfile.

3. <u>END1</u>

Job termination is handled in this routine. If the job terminates normally via an operator drop, the following procedure is executed.

1. Drop the CP
2. Get address of FET from RA+4
3. Set up FET function code for close reel
4. Call CLO for closing procedure
5. Store END in RA+1
6. Drop PP

<u>General Flow</u>

The FNT is searched for a file (SCFNT), and if initially no files of the type specified are found, the PP is dropped. The FNT channel is requested and the FNT entry is written to CM (RCH). The FNT is searched again to save search time the next time and the address of the next FNT entry is saved until needed. If the CM program requests an entry and none exists, the PP loops searching the FNT and pausing for relocation. If the error flag is set, termination procedure is executed (END1).

XRQ

Called from DSD

XRQ is loaded in the same manner as XDQ, and XXXRESQ is loaded in the same manner as XXXDMPQ.

Called from XXXRESQ

XRQ is called from XXXRESQ with the address of a 4-word buffer in bit 0-17 of the call. As the address will be greater than 21B, XRQ assumes that this is not the initial DSD call. The first location of the buffer is monitored by XRQ and if it is zero, the routine pauses for relocation. If the cell is nonzero, a check for 7777B is made, if yes, the pp routine drops the CP and PP. If not equal to 7777B, XRQ assumes that it is the address of the FNT of the file being restored. The next three words of the buffer is read into the PP. These three words are the actual FNT entry of the file before it was dumped. XRQ checks if the file is type output and if it is the backspace bit is checked. If the bit is on, the current RBT word pair is advanced the proper number of RBT bytes. The FNT entry is then written to CMR and the PP again enters into its main wait loop. As soon as the PP reads the FNT entry from the CM program, it sets the location to zero so the CM routine can continue processing.

XRQ

Called from DSD

XRQ is loaded in the same manner as XDQ, and XXXRESQ is loaded in the same manner as XXXDMPQ.

Called from XXXRESQ

XRQ is called from XXXRESQ with the address of a 4-word buffer in bit 0-17 of the call. As the address will be greater than 21B, XRQ assumes that this is not the initial DSD call. The first location of the buffer is monitored by XRQ and if it is zero, the routine pauses for relocation. If the cell is nonzero, a check for 7777B is made, if yes, the pp routine drops the CP and PP. If not equal to 7777B, XRQ assumes that it is the address of the FNT of the file being restored. The next three words of the buffer is read into the PP. These three words are the actual FNT entry of the file before it was dumped. XRQ checks if the file is type output and if it is the backspace bit is checked. If the bit is on, the current RBT word pair is advanced the proper number of RBT bytes. The FNT entry is then written to CMR and the PP again enters into its main wait loop. As soon as the PP reads the FNT entry from the CM program, it sets the location to zero so the CM routine can continue processing.

1BT <u>Blank Label Routine</u> (For Tapes and Disk Packs)

1BT is called by DSD to a vacant control point following one of two type-ins:

n.BLANK. To write blank labels on degaussed or blank tapes. The blank labels written are the standard SCOPE 3.0 tape labels, viz. a volume header label (VOL1), a file header label (HDR1) and a tape mark, in that order. The operator will be requested to assign tapes to be labelled and to enter a visual reel number which will be written in the volume header label.

n.BLANK,uu. where uu is the EST ordinal of a disk pack unit; to write a blank label on a disk pack. The label is written in the 64 CM words of PRU 0, record block 0, of the disk pack; the 640 characters are written as follows:

1 - 4 "DEV1"

5 - 10 the right 6 characters of CMR word T.JDATE: 00B followed by the five-character Julian date.

11 - 78 binary zero.

79 - 80 a 12-bit checksum of all the other 2-character bytes in the label.

81 - 460 a skeleton RBR entry (38 CM words) for an empty public disk pack, viz.:

```
0700 0000 0000 0301 0100
3720 0000 3717 3717 0000
4000 0000 0000 0000 0000 0000
...30 zero words...
0000 3777 7777 7777 7777 7777
... 4 zero words...
```

which shows that the first record block is occupied by the label, the next 3717B are both physically and logically available, and 47 are physically non - existent.

461-640 binary zero.

The tape and disk functions are not really related, but are brought together in 1BT for convenience.

When 1BT is called, bits 0-11 of the input register contain either 0, indicating the tape function, or the EST ordinal of the disk pack unit. Subroutine PRS sets the job name "BLANK", sends it to the dayfile, and gets a field length of 1000B words, priority 7700B, and time limit 1000B.

Tape Function

If bits 0-11 of the input register do not contain 0, we branch to location DISK. Otherwise, proceed with the tape function. 1BT prepares an FET in central memory, the first two words of which it also uses as REQUEST function parameters. The FET is 13 words long so that it can hold label information and so that the FNT pointer will be filled in by a CIO request. The only field in the label portion of the FET that is set is the Retention Cycle field. This is set to display coded zeroes so that the tape on which the blank label is written is regarded as expired.

REQUEST TAPE and REWIND TAPE functions are accomplished by asking monitor to call the REQ and CIO programs into other PP's. These functions are called sequentially with the FET being modified as necessary for each call. The FET is monitored for completion of operation after each call.

4LB, the labelled tape driver, is called as an overlay to 1BT three times per blank tape to be labelled, once for each of the parts of the label. CIO is called after the labels have been written to rewind the tape again.

2DF is called as an overlay to drop the file from the FNT and the process repeats with a REQUEST TAPE as mentioned above.

Disk Function

At DISK, we begin by reading the EST entry for the device specified by bits 0-11 of the input register. If this EST ordinal is out of range, we go to DERR1 to issue the dayfile message "EST ORDINAL OUT OF RANGE", and then wait at DERRB till the operator drops the control point. If the EST entry is not for a disk pack that is logically on the system, we go to DERR2 and do the same with the message "NOT DISK PACK, OR OFF". If the status of the disk pack is not unloaded (4040B), we go to DERR2 and do the same with the message "PACK STATUS NOT UNLOADED". Now call monitor to assign the pack as private at this control point. If it is not granted, go to DERR4 and do the same with the message "CANT GET PACK ASSIGNED". If all is well so far, fill the EST ordinal into the stack request at MYRQ ff., which has been pre-assembled to write bytes 4001B through 4500B of PP memory to PRU 0, RB 0, of the device, with no FET or FNT.

Next construct a blank label in this area, first zeroing it; then inserting "DEV1", the Julian date, and the skeleton RBR table, and forming the checksum. Note the use of subroutine COPY, which is entered with address x in D.Z1 and y+z*10000B in the A register; it copies bytes y through y+z-1 into x through x+z-1. Next load overlay 5DA at 5000B ff., and enter it with 5 in the A register to indicate the BLANK function. If 5DA exits with zero in the A register, the disk pack does not contain a private pack label, and we branch to DISKB to write the blank label. First add to the stack request at MYRQ ff. by inserting this PP's message buffer address; then scan the RBR tables in CMR for the one containing the proper EST ordinal. If this cannot be found, go to DERR5, issue the

dayfile message "NO RBR TABLE FOR PACK", and wait at DERRB until the operator drops the control point. If the right RBR table is found, copy the DST ordinal from it into the stack request, and call R.WRITEP to execute it, thus writing the blank label on the pack. Format the EST ordinal into the message at GOODMES, "uu BLANKED." and send it to the dayfile. Then request monitor to return the disk pack to unloaded status. If this succeeds, go to EXIT to drop the PP and the control point. If not, go to DERR6, send the message "CANT UNLOAD PACK AFTERWARDS" to the dayfile, and wait at DERRB till the operator drops the control point.

If, on return from 5DA, the A register contains non-zero, the disk pack appeared to contain a private label, and the A register contains the starting address of a copy of its first PRU, which has been read into the PP. We now copy the pack name, Julian date, and visual identification from this copy into the message at DMSG7, "PN = pname, VN = vrno, JD = jdate", and send this message to the dayfile. The same message appears on the first line of the B-display, and we copy to the second line the message "PAUSING FOR GO OR DROP" (at DMSG8). Now set bit 12 of RA+0 to 1 and wait until the operator types "n.DROP." or "n.GO.". If the operator wants to save the private-labelled pack, he types "n.DROP.", and on seeing the error flag we go to EXIT to drop the PP and the control point. The pack will be returned to unloaded status by 1EJ as part of the termination process. If the operator still wants to blank label the pack, he types "n.GO.", which causes bit 12 of RA+0 to be reset to 0. On seeing this, we continue to DISKB to write the blank label and so on as if the disk pack had not been privately labelled in the first place.

Entry Information

For blank labelling tape, bits 0-11 of the input register contain 0, and one or more tapes are specified by the operator through REQUEST - ASSIGN sequences. For blank labelling a disk pack, bits 0-11 of the input register contain the EST ordinal of the pack.

Exit Information

None except the messages.

Other Programs Called

5DA, for disk pack but not for tape. The return from 5DA is with 0 in the A register if the pack is not privately labelled, otherwise with the address of a copy of the first PRU of the private label.

4LB and 2DF for tape but not for disk pack.

dayfile message "NO RBR TABLE FOR PACK", and wait at DERR8 until the operator drops the control point. If the right RBR table is found, copy the DST ordinal from it into the stack request, and call R.WRITEP to execute it, thus writing the blank label on the pack. Format the EST ordinal into the message at GOODMS6, "uu BLANKED." and send it to the dayfile. Then request monitor to return the disk pack to unloaded status. If this succeeds, go to EXIT to drop the PP and the control point. If not, go to DERR6, send the message "CANT UNLOAD PACK AFTERWARDS" to the dayfile, and wait at DERR8 till the operator drops the control point.

If, on return from SDA, the A register contains non-zero, the disk pack appeared to contain a private label, and the A register contains the starting address of a copy of its first PRU, which has been read into the PP. We now copy the pack name, Julian date, and visual identification from this copy into the message at DMSG7, "PN = pname, VN = vrno, JD = jdate", and send this message to the dayfile. The same message appears on the first line of the B-display, and we copy to the second line the message "PAUSING FOR GO OR DROP" (at DMSG8). Now set bit 12 of RA+0 to 1 and wait until the operator types "n.DROP." or "n.GO.". If the operator wants to save the private-labelled pack, he types "n.DROP.", and on seeing the error flag we go to EXIT to drop the PP and the control point. The pack will be returned to unloaded status by 1RJ as part of the termination process. If the operator still wants to blank label the pack, he types "n.GO.", which causes bit 12 of RA+0 to be reset to 0. On seeing this, we continue to DISKN to write the blank label and so on as if the disk pack had not been privately labelled in the first place.

Entry Information

For blank labelling tape, bits 0-11 of the input register contain 0, and one or more tapes are specified by the operator through REQUEST - ASSIGN sequences. For blank labelling a disk pack, bits 0-11 of the input register contain the EST ordinal of the pack.

Exit Information

None except the messages.

Other Programs Called

SDA, for disk pack but not for tape. The return from SDA is with 0 in the A register if the pack is not privately labelled, otherwise with the address of a copy of the first PRU of the private label.

4LB and 2DF for tape but not for disk pack.

1DF Dump Dayfile

1DF is called by 1EE following the keyboard entry n.DAYFILE,uu. The parameter uu indicates the media the system dayfile is to be dumped to. It may assume the values LP, CP, or MT. 1DF expects one of these three arguments in byte 4 of its input register. Upon being called 1DF sets its direct cells and requests a field length of 40000B, a priority of 7700B, and a time limit of 1000B. 1DF then jumps to the proper routine to process the assigned equipment type. If the equipment is not of the proper type the message ILLEGAL EQUIPMENT REQUEST is issued and the job dropped.

All three equipment types use the subroutine CDF (Complete Dayfile). CDF issues the MTR function M.CDF to complete the dayfile and issues the message DAYFILE DUMPED. CDF then rewinds the system file DAYFILE and creates a new file local to 1DFs control point, also by the name of DAYFILE and with the proper disposition for the requested device. All RB assignments associated with the system file DAYFILE are transferred to this newly created file. When the dump process is completed this new file will automatically be dropped, releasing the reserved record blocks.

In the case of the two equipment types LP or CP (line printer or card punch) all that is necessary is the creation of a file with the proper disposition. The job is dropped when 1DF releases its PP and the printing or punching of the dayfile will be handled by the output package. In the case of equipment type MT (Magnetic tape) it is necessary to copy the local file DAYFILE from disc to tape. 1BT does this by placing three control statements in the control statement buffer for its control point. These three statements are REQUEST TAPE, COPYBCD (DAYFILE, TAPE), and BKSP (TAPE). When executed these statements will cause the dayfile to be copied to tape and the tape backspaced over the trailer information to reposition it for the next dayfile dump. After setting up the control statement buffer 1DF exits, allowing the system to handle the processing of the statements as it would for a normal job. Upon completion of the statement processing the job will be dropped and along with it the local file DAYFILE.

1MF **MULTIFILE POSITIONING**

Refer to Chapter 11, page 11-1.

1MR OPEN Read/Alter Magnetic Tape File

Refer to Chapter 11, page 11-1.

1MW <u>OPEN Write Magnetic Tape File</u>

Refer to Chapter 11, page 11-1.

1PL--Dummy Plot Program

General

1PL is a PP program, always loaded at 1000B. It is called only by JANUS for OUTPUT files having a PLOT (0030) Disposition Code.

Function

JANUS calls 1PL to simulate a system Plotter Driver for OUTPUT files having a PLOT Disposition Code.

Entry Information

The input register contains:
VFD 18/342014B, 6/a, 18/0, 18/b where a is the Control Point number and b is the FET address relative to RA.

Narrative

1PL checks the control point error flag and if it is set drops the PP. If the control point error flag is not set, then 1PL sets byte 0 of word zero of the FET to 7777B and drops its PP.

Subroutines

None.

Exit Information

None.

1RI - ROLL-IN

GENERAL

1RI is a PP program loaded at 1000B and called in by DSD as a result of an n.ROLLIN. type-in at the console.

FUNCTION

1RI complements the functions performed by 1RO. Specifically, 1RI reinstates a program that was previously rolled out and reactivates the program by restoring its central processor status. The rolled-in program remains at the control point with which it was associated when it was rolled out.

ENTRY INFORMATION

When 1RI is initially called in by DSD, a file named ▿ωROLOUTn▿ where n is the control point number and the programs field length, C.P. status and file FNT address all of which are in the programs RA+10. If 1RI is recalled {1RI enters the delay stack at control point zero if there is insufficient C.M. available to rollin the program} the FNT address, the control point address and the programs FL, all this information is in the input register bytes 2,3, and 4.

EXIT INFORMATION

None.

OTHER PROGRAMS CALLED

None.

NARRATIVE

1RI checks the control point number in the input register. If it is zero, then 1RI was in recall, therefore the programs FL, CPA address and the file's FNT address are all in the input register. A request for CM is made, if granted 1RI

transfers to read in the program, otherwise 1RI goes into recall at control point zero.

If the control point number in the input register is non-zero, the FNT entry is scanned for the file named ▽@ROLOUTn▽ (where n is the control point number) and if the file is there continued. Otherwise a message ▽CIO ERROR 1▽ followed by ▽ROLLIN ABORTED▽ is written to the dayfile, the error flag in the CPA is set =3 and 1RI drops out.

When the FNT entry has been checked the status in the associated FST is tested and if the status is busy error messages ect., as above, are written to the dayfile and 1RT drops.

If 1RI is still in business then an attempt is made to ensure that there is no activity at this control point, if this is impossible ▽ROLLIN ABORTED▽ is written to the dayfile and 1RI drops.

1RI then reads in the program's RA+10 where the FL was stored, displays ▽ROLLIN INITIATED▽ message, saves the FL, CPA address and FNT entry address in the input register and request CM equal to the field length. If the request is honored 1RI continues otherwise 1RI sets itself in delay at control point zero and then drops.

The program can now be read in. First, 11 CM words saved at RA+10 are read in, these include the programs CP status, FL, last dayfile message buffer and last two words of CM field. The DST ordinal is then located and stored in the stack request, the request is made and 1RI waits for I/O completion. If there is an I/O error the messages ▽CIO ERROR 3▽ and ▽ROLLIN ABORTED▽ are written to the dayfile and 1RI drops. Otherwise 1RI stores the 1st RBT word pair address in the stack request, zeros the FNT entry, transfers to stack request to drop file ▽@ROLOUTn▽, restores the programs last dayfile buffer, last two CM field words and the central processor status, updates the rollout flag, decrements the psuedo-activity count in MTR and exits.

SUBROUTINES

PPACT

Tests for control point activity, if none returns otherwise will retest. The activity test is made for 2 seconds and if at the end of that time the control point is still active rollin is abandoned, the rollout flag is reset to job rolled out (FLAG = 3), ≠ROLLIN ABORTED≠ is written to the dayfile, and 1RI drops. The number of times the activity loop is executed can be varied by changing the initial values of either CNTR or CNTR+1 in this subroutine.

DISMSG

This subroutine writes a message in the last dayfile message buffer for console display, and returns.

1RO - ROLL-OUT

GENERAL

1RO is a PP program loaded at 1000B and called in by DSD as a result of an n.ROLLOUT. type-in.

FUNCTION

A job running in CM is utilizing resources which may be needed by some other job, in particular another job may be waiting for central memory. If the resources used by the first job are to be allocated to the second then the first job must be rolled out. This is accomplished by 1RO. 1RO will write the job to be rolled out to a file named ▽@ROLOUTn▽, where n is the control point number at which the job is running, after deactivating this job and will relinquish all but 100 CM words previously assigned to the job. In fact, after a job has been rolled out the only resources which remain associated with the job are a control point and 100 CM words.

ENTRY INFORMATION

None.

EXIT INFORMATION

The rolled out job's FL, the FNT address of ▽@ROLOUTn▽ and CP status as well as its last dayfile buffer and last two FL words are saved in RA+10 to RA+20.

OTHER PROGRAMS CALLED

None.

NARRATIVE

1RO checks the CPA error flag the jobs field length and the jobs priority. If the error flag is non-zero or the field

length less than or equal to 100 or the priority is zero 1R0 drops. If none of these conditions hold then 1R0 tests for activity at this control point by dropping the central processor and requesting from Monitor an activity count. If the control point is inactive, i.e. the only activity is the PP used by 1R0, then 1R0 continues. Otherwise it will retest the activity until either the control point becomes inactive in which case 1R0 continues or the test has been performed for a period of 2 seconds in which case 1R0 displays "ROLLOUT ABONDONED", restores the central processor, if necessary, resets the rollout flag and drops.

When the control point becomes inactive a search through the FNT is made for a file named "@ROLOUTn" and a free entry. If either such a file is located or there is no free entry 1R0 drops otherwise "@ROLOUTn" is inserted in the FNT at the free entry. "ROLLOUT INITIATED" message is then displayed and the jobs FL-2 is written out. When the I/O completes the file is reqound. If the I/O terminated with an error "ROLLOUT ABORTED" is written to the dayfile, the rollout flag is reset to job not rolled out then 1R0 drops.

Otherwise the jobs FL, CP status, last dayfile buffer, last two CM words and the FNT address of "IROLOUTn" are saved at RA+10 through RA+20, the rollout flag is set to job rolled out and 1R0 drops after requesting anFL of 100 CM words, and increments the psuedo-activity count via MTR.

SUBROUTINES

DROPCP

This subroutines drops the central processor and saves the status returned by the monitor function M.DCP.

MS9DIS

This subroutine writes a message in the jobs last dayfile message buffer for console display.

1TD Tape Dump

This routine is used to dump print files to tape. It is loaded when the operator types in X.DUMP. The logic involved is to search the file name tape for a print file and put the name of this file into a statement of the form COPYBCD (XXXXXXX,TAPE) (the file name replaces the XXXXXXX). Then the statement is written to the control statement buffer of the control point, and 1AJ is called in to process the control statement. The result is that the print file gets copied to tape. When there are no more output files in the system, 1TD releases its PP and exits to its idle loop.

The main routine of 1TD merely reads the input register, stores the control point address, presets some constants, and calls first the subroutines CPS and REQ and then loops, calling the subroutines SCH and DUMP until no further print files remain.

The subroutines of 1TD and their functions will be described below.

CPS - Enter control point status

This routine is used to set up the control point area constants and initialize some other values. Since 1TD comes in and out of recall at one point in its execution, it is first necessary to check to see if 1TD has been at this control point previously or not (i.e., whether it has been loaded initially or is just coming back out of recall). This information is derived by checking the job name. If it is non-zero, then 1TD is just coming back out of recall and the code in CPS is not necessary. In this case CPS is exited. If 1TD is in for the first time, it requests storage and a time limit from MTR. If the storage is not assigned, the PP is dropped and 1TD terminates. If it is, RA and FL are recorded and (RA) - (RA+7) are cleared.

REQ - Request tape

This routine is used to request a tape from the operator. It just checks the job name to see if this is the first time in. If so, it enters the job name, DUMP, and sends the statement, REQUEST (TAPE), to the control statement buffer, calling in 1AJ to process it. If 1TD was merely coming back in out of recall, the above is not done. In either case, it now searches the FNT to see if a file has been assigned. If so, its FST address is saved and 1TD exits from REQ. If no file has been assigned, 1TD sets a delay of about 2 seconds and goes into recall.

SCH - Search for output file

This routine looks for an output file in the FNT. It requests the FNT channel and looks through the FNT for an output file of print disposition and, finding one, saves its priority for comparison against the highest current priority. At the end of the search the print file with highest priority is selected to be dumped. If no print file remains in the FNT, the FNT channel is dropped, the PP is dropped and 1TD terminates. If a file is found, it is assigned to this control point as a local file, its priority and disposition code are cleared, the FNT channel is dropped, and the file name is inserted as the job name of the control point. 1TD then exits from SCH.

DUMP - Dump file to tape

This routine first merges the file name into the COPYBCD statement and then calls RNS to write the statement to the control statement buffer. It then loops, looking at the FST status word to see if it reflects a completed EOF. When it does, DUMP calls RNS again to put the statement BKSP (TAPE) in the statement buffer and DUMP then loops until the FST status reflects a completed skip-backward. At this time 1TD exits from DUMP.

RNS - Enter control statement

This routine clears the control statement buffer, sets the control statement pointer to the beginning of the buffer, and stores the designated statement in the buffer (the address of this statement is in the A-register upon entry to RNS). It then uses CALL1AJ to bring in 1AJ to process the statement. Upon return from CALL1AJ, 1TD exits from RNS.

CALL1AJ - Bring in 1AJ

This routine asks MTR to bring in 1AJ at this control point. The monitor function M.RPJ (request peripheral job) is used. Upon return from R.MTR, 1TD exits from CALL1AJ.

CHECKEF - Check error flag

This routine is called periodically in 1TD to see if an error has occured. It reads the error flag from the control point area and checks to see if it is set. If not, 1TD exits from CHECKEF. If the flag is set, 1TD releases the PP, jumps to the idle loop, and terminates.

2BP - CHECK BUFFER PARAMETERS

Introduction

When action is intended for a file, 2BP can be used to validate the buffer parameters of the File Environment Tape (FET). It also locates or creates an associated File Name Table (FNT/FST) entry.

Entry Information

When calling 2BP the following information must be given:

D.PP IRB+C.CPNUM = Right justified control point assignment

D.PP IRB+3 and D.PPIRB+4 = Relative CM address of FET to be processed

D.RA = Control point RA (hundreds)

D.FL = Control point FL (hundreds)

This information can be prepared by using the system macro PPENTRY with D.PPIRB as the first parameter.

General

The FET name is checked to be no more than seven alpha-numeric characters. The first character must be alphabetic. The buffer parameters are checked to be in the following range:

```
FET(1) +4+1  FL
LIMIT  FL
OUT  LIMIT
IN  LIMIT
OUT  FIRST
IN  FIRST
```

The validated name in the FET and the Control Point assignment in the input register are used to search the FNT for an existing entry. If no entry is found, an entry is made in the first empty space found in the FNT. This new entry has a zero device type.

If an FNT entry must be created and there is no available space, normal processing will not continue. If the l field in the FET is zero, an appropriate dayfile message is written and the control point is aborted. If the l field is non-zero, 24 octal is returned in bits 9-13 of the code and status field of the FNT.

The requested code and status field is transferred from the FET to the established FNT after the last buffer status in the FNT has been placed in direct location BS. The device type in the established FNT is placed in the FET and direct loaction D.DT.

If the high order bit in the CIB Link field of the FNT is off, the FET address is stored in this field.

Error Procedure

When an invalid file name or an illegal buffer parameter is detected in the FET, an appropriate dayfile message is written and the control point is aborted.

If 1 is non-zero in the FET, the disposition code from the FNT and the FNT address will be placed in the FET.

Exit Information

The following information is prepared and left by 2BP:

D.FNT to D.FNT+4	= 2nd word of FNT
D.FNT+5 to D.FNT+9	= 3rd word of FNT
D.DTS	= device type found in FNT (zero if FNT created by 2BP)
D.BA to D.BA+4	= 1st word of FET
BS = D.BA+5	= last buffer status found in FNT
FA = D.FL+1	= address of 2nd FNT word
D.FIRST to D.FIRST+1	= "FIRST" parameter of FET
D.IN to D.IN+1	= "IN" parameter of FET
D.OUT to D.OUT+1	= "OUT" parameter of FET
D.LIMIT to D.LIMIT+1	= "LIMIT" parameter of FET

2DF - DROP FILE(S)

General Description

2DF is called as an overlay by PP programs which wish to remove one or more files from a control point. 2DF removes files by releasing common files (assigning them to control point zero), dropping local files and locking private disk pack files. If the error flag is set to F.ERRN(RERUN) the common file change bit is ignored in determining which files are common and which are local.

Entry Information

The value of direct cell D.BA determines what action 2DF is to take:

D.BA = 1	2DF is to remove all files assigned to the control point.
D.BA = 0	2DF is to remove a single file. The address of the files FNT is in D.BA+1.
D.BA = NAME	If D.BA contains neither a zero nor a one, then 2DF is to drop a single file whose FNT entry is in D.BA - D.BA+4. That is, D.BA - D.BA+3 contain a 7 character file name, and the lower 3 bits of D.BA+3 contain a control point number. 2DF is to search the FNT until it find a file of the given name at the given control point and then remove the file. If no such file is found 2DF exits.

Exit Information

If the FNT for the (or a) specified file is zeroed the direct cell D.FA is set to zero. Otherwise D.FA is not changed.

2LP

Function

2LP is a PP program, always loaded at 2000B. It is called only by CIO, when CIO finds itself writing a local file assigned to a 501 or 512 printer (EST mnemonics LP and LQ.) 2LP reads data from the circular buffer and sends it to the printer.

Entry Information

D.EST through D.EST+4 contain the EST entry for the printer.

D.FNT through D.FNT+9 contain the second and third words of the FNT entry for the local file to which the printer is assigned. Special use is made of 6 bits in D.FNT+C.FLBL (D.FNT+3) as follows: Bit 6 (FL.NONXT=100B) is 1 if the first character of the next line (if any) cannot be found or cannot be obeyed, if it calls for a pre-print skip. Whenever possible, a pre-print skip is handled as a post-print skip for the preceding line.

Bit 11 (FL.N2NXT=4000B) is set to 0 at the beginning of a line unless bit 6 is 1; in that case bit 6 is reset to 0 and bit 11 is set to 1. This means that if the first character of the present line calls for a pre-print skip, the skip has not already been carried out, and it will be necessary to skip before printing this line. (To get the effect of a pre-print skip, a minimum-length blank line will be printed with the corresponding post-print skip.)

Bit 7 (FL.POST=200B) is set to 1 whenever a post-print skip function is senc to a 501 printer. After the line is actually printed, this flag is seen and reset to 0, and a 10B (clear post-print skip) is sent to the printer. A flag in the FNT is used to signal this, rather than a cell in the PP program, although there does not seem to be any way 2LP could be dropped and re-loaded between the two events.

Bit 8 (FL.CUT=400B) is set to 1 whenever the current line is terminated because it appears to be longer than 155 bytes. Its continuation is printed as a separate line, and at that time bit 8 will signal that the first character of the new line is to be ignored rather than treated as a format character.

Bit 9 (FL.AUTO=1000B) is set to 1 whenever a line begins with an ▽R▽ format character, to indicate automatic skipping from the bottom of each page to the top of the next. It is reset to 0 whenever a line begins with a ▽Q▽ format

character. As long as this bit is 1, a 5 function is sent to the printer before printing each line.

Bit 10 (FL.8LL=2000B) is set to 1 whenever, using a 512 printer, a line begins with a "T" format character, to indicate 8 lines per inch spacing. It is reset to 0 whenever a line begins with an "S" format character, to indicate a return to 6-line spacing. The 8-line function is sent to the printer when bit 10 is first set, and thereafter whenever 2LP is reloaded, if bit 10 is 1, and whenever, bit 10 being 1, a line with a "Q" format character is processed. "Q" calls for a termination of automatic page skipping, and the clear function sent to the printer at this time also clears the 8-line status, so the latter has to be restored if bit 10 is 1.

D.PPIRB through D.PPIRB+4 contain a copy of the CIO request, with the address of the FET, relative to RA, in bits 0-17.

D.IN, D.IN+1, D.OUT, D.OUT+1, D.FIRST, D.FIRST+1, D.LIMIT, and D.LIMIT+1 hold the current values of the FET pointers.

Exit Information

Bits 6 through 11 of D.FNT+C.FLBL contain information as described under Entry Information.

D.OUT and D.OUT+1 now equal D.IN and D.IN+1. However, the OUT pointer in the FET has been updated only to the last +1 word actually printed; or if a record has just been completed, the IN and OUT pointers in both PP memory and the FET have been set equal to the FIRST pointer.

Other Programs Called - none

Messages

If any print line begins with the characters "PM", it is not printed, but is sent to the dayfile as a message, and appears in the control point B-display. The program waits until the operator types "n.GO."

PRINT ERROR   This message goes to the dayfile and the B-display when a 512 printer sets bit 2 or bit 10 in its status byte (Compare Fault and Print Error) to 1. The program waits until the operator types "n.GO."; then re-initializes the printer and continues printing the file.

NOT READY   This message appears on the B-display as long as the printer status is not ready (0 in bit 0).

RESERVED   This message appears on the B-display as long as the 501 printer is reserved by another channel than the one 2LP is using to check its status. Since SCOPE requires that the printer be assigned exclusively to a control point before its status is checked, this condition could only arise if the printer were connected to two different computers.

REJECT   This message appears on the B-display as long as the 6681 status shows that a function code has been rejected (bit 1 or 3 is 1).

XMSN PARITY ERROR   This is put on the B-display and the dayfile when the 6681 status indicates a transmission parity error (bit 0 is 1). The program then waits one second, issues a clear channel function, and tries again.

These messages are all preceded by ≠LP≠ and the EST ordinal of the printer.

Narrative

To save a little space, the initializing section of 2LP is located in cells afterward mostly used for the print line image. On entering 2LP, we immediately branch to PRS, check the status byte of the FNT, and set cell EOR+1 to zero if not end-of-record, or non-zero if end-of-record. Then zero cell PRSY, so that if this code is executed a second time, we shall immediately branch to LPDA. If 2LP is entered more than once without being re-loaded each time, the only presetting that has to be done after the first entry is of EOR+1. So the print line image area can begin at BUF, the cell next after the branch to LPDA.

On the first entry to 2LP, we do not branch immediately to LPDA, but get the EST ordinal out of the FNT entry, format it, and insert it after the letters ≠LP≠ in the message area beginning at MSGA. Then determine, from the EST entry, whether the printer is a 501 or a 512. If a 501, we zero a number of program switches, thus setting them to NOP instructions; and also zero one or two constants so as to adjust the program from 512 to 501 operation. Then branch to FILLT, call subroutine LOAD, and go to LPDD to start processing the first line. Subroutine LOAD, for a 501, merely tests bit 21 of the second word of the FNT entry to see if the file is in auto-page-skip status, and if so passes the corresponding function code, 5, to the printer. This may not be necessary, but as 2LP has no way of knowing who may have tinkered with the printer, it is done anyway.

For a 512, we leave the program switches set as they were assembled, and set up the printer train image. Only the standard 64-character image is provided, whereas in the 512-driver part of 1IR-1IS, a 48-character image and an interface for a 288-character image are also provided. Between PRSB and FILLK, by a procedure needing no clarification, the array of 288 characters including 63 different ones is set up in 144 bytes beginning at IMAGE. From FILLK on, each of these bytes is expanded into two bytes, padded with display code zero as their left halves, giving an array of 288 bytes beginning at IMAGE. This expansion is done from right to left, as the extra space for it is taken on the right. Converting this array from display code to BCD will give what has to be copied into the printer's train image buffer; display code zero has been inserted into the left half of each byte because the BCD code for the character is 00B. Finally we call subroutine LOAD. For a 512 printer, LOAD loads the array into the train image buffer, and then refreshes auto-page-skip status if necessary, as above for the 501. Finally we branch to LPDD. Note that IMAGE through IMAGE+287 must remain undisturbed in case LOAD is called again after a print error. The print line image area is below IMAGE.

For the first print line it processes after being called, 2LP begins work at LPDD; for all subsequent lines, at LPDA. The reason for the difference is that the code between LPDA and LPDD is to be executed once after each line, and not twice. If 2LP has just been called for a new file, this code need not be executed. Otherwise, the last time 2LP worked on this file, it executed the code after printing the last line, and before discovering that there wasn't another line left in the buffer. So if 2LP initially began at LPDA, this code would be executed a second time after the printing of only one line.

At LPDA we begin work on a line by setting flag LF.N2NXT to 1 if FL.NONXT=1 and then setting FL.NONXT to 0; otherwise by clearing them both to 0. FL.NONXT may have been set during the processing of the preceding line to indicate that the first character of this line, if it calls for a pre-print skip, has not been obeyed as it normally would have been. We are now setting FL.N2NXT if necessary, so as to take the proper action for this line a bit later, and clearing FL.NONXT so that it becomes immediately available to carry information about the next line. (If this promotion of FL.NONXT to FL.N2NXT were done twice between lines, the information would vanish.)

At LPDD we set WC, the byte counter for the line, to 0, and call subroutine XFR to get the next (first) word out of the circular buffer into the next available space in the line image buffer, which begins at BUF.

If the return from XFR is with non-zero in the A register, a word was taken from the buffer and we branch to LPD11. If the return is with zero, the buffer was already empty, and so we have not succeeded in beginning a line. There is nothing to do but exit from LPD; if the FNT status was end-of-record when 2LP was entered, we set IN= OUT= FIRST in the FET before exiting 2LP.

At LPD11 we check the last byte of the last CM word moved into the print line image area; if this is 0000B, it terminates the line image and we go to KA. At KA, we would like to look at the first character of the next line image, as it may affect the format of this line. If IN does not =OUT, we can look at it, so at KAF we read the next word, copy its first character into NCHAR, and so arrive at KAG. But if IN=OUT we cannot see the next character; we go to KAC, set FL.NONXT=1 to show that it will have to be handled later if it calls for a pre-print skip; then set NCHAR to zero and so reach KAG.

If the line image is not found at LPD11 to be terminated, we test the byte count, in WC. If this has reached 155 we shall arbitrarily cut off the line, and we branch to KAA to do this. If not, we call subroutine XFR to get the next CM word in the line; if the return is with non-zero in the A register, the word was obtained and we return to LPD11. If with zero, the buffer was empty and no word was obtained. If the FNT status was end-of-record on entry to 2LP, 2LP assumes that it was meant to print out the rest of the buffer even though there was no terminator, so we go to KAC. {This case, buffer exhausted, end of record, no normal terminator, is treated the same as buffer exhausted at the end of a line with a normal terminator; i.e., e.o.r. is allowed to serve instead of a zero byte at the end of the last CM word in the record, which seems reasonable.} If the FNT status was not end-of-record when 2LP was entered, and the buffer is exhausted before we see a normal terminator or reach a byte count of 155, we simply exit from 2LP, leaving the OUT pointer in the FET still pointing to the beginning of the uncompleted line. Either some other program will put more of the record into the buffer, or it will call for printing the incomplete last line by calling 2LP to write end of record.

We go to KAA if the line image becomes 155 bytes long before a terminator is found. If IN=OUT, so that the buffer is exhausted, and the FNT status when 2LP was called was end-of-record, we go to KAC. In other words, buffer exhaustion with end of record but no normal terminator is handled the same when the image is 155 bytes long as when it is shorter. But if the buffer is not exhausted,

or the FNT status was not end-of-record, this is probably an artificial cut we are going to make after the 155th byte, and we arrive at KAD; set CUTFLAG to 1, then set NCHAR to 0 and go to KAG. It would be logical to set FL.NONXT to 1 in this case; but the first character of the next line image is going to be disregarded altogether, so there is no need to set the flag that shows it has not been obeyed in case it calls for a pre-print skip.

When, one way or another, we have terminated a line image and set NCHAR to contain either the first character of the next line or zero, we arrive at KAG. Now if FL.CUT = 1, the end of the preceding line must have been arbitrarily determined, so the first character of this line is not likely to have been intended as a format character. We reset FL.CUT; then go to LPR2 to set FORMAT to zero rather than the first character of this line, and then print the line. If FL.CUT is 0, we have to consider the first character of this line. If the first two characters are ▽PM▽, the line is intended to appear on the dayfile and the B-display, rather than being printed. If not, branch to LPR to print it. But if so, set FL.NONXT=1, indicating that the first character of the next line, if it calls for a pre-print skip, has certainly not been obeyed during theprocessing of this line. Then blank the ▽PM▽. As the dayfile message is not to be more than 15 bytes long, including the zero byte at the end, we zero BUF+14, to terminate it by force if already longer than that. Then if the byte count in WC is not less than 14, we go to PSE2 to send the message to the dayfile. If it is less than 14, we zero bytes following the last one up to and including BUF+13 and then go to PSE2. At PSE2, send the line to the dayfile and the B-display; then call subroutine WAIT to wait until the operator types ▽n.GO.▽, and so reach LPRW.

We come to LPRW after printing or displaying a line. If CUTFLAG is nonzero, the line was terminated arbitrarily; we zero CUTFLAG and then set FL.CUT to 1; the use of this in the next line is explained at the beginning of the preceding paragraph. At LPD7A, we update the OUT pointer in the FET to reflect that the line has been definitively taken from the buffer and disposed of. Unless 2LP was called with end-of-record status in the FNT, in which case the IN pointer then existing must be taken as final, we update D.IN and D.IN+1 according to the FET, and return to LPDA to begin the next line.

If a line is to be printed rather than displayed as a dayfile message, we come to LPR or LPR2 and put either the first character of the line or zero into FORMAT, depending on whether FL.CUT was 0 or 1. Since the first character

is no longer wanted, we shift the whole line one character to the left, so that cell BUF contains the second and third characters of the original image. At LPRE, we begin to scan the line from right to left, discarding one byte after another by reducing WC, until we come to a byte that does not contain two blanks (0000B, 0055B, 5500B, or 5555B). However, if the entire line turns out to be blank, we keep the count in WC at 1.

At LPRF we begin working on the format of the line. We test FL.N2NXT, and pass on to LPRJ if it is 0. If it is 1, it means we have not taken care, during the preceding line, of a possible pre-print skip called for by the first character of this line. Then we put the content of FORMAT in the A register, the first character of this line or zero, and call subroutine DUM to do pre-print skip if necessary. This does not conflict with later format processing, as DUM acts on exactly those format characters that are not checked for in the code between LPRJ and SPCM. So whether DUM is called or not, we arrive at LPRJ.

If the character in FORMAT is 00B, we go straight to SPCG. If this were not done, the character would fail all the tests and we would arrive at SPCG just the same. Note that a blank format character will fail them all and send us to SPCG.

If FORMAT contains Q, this calls for cancelling automatic skipping from bottom to top of page. We set FL.AUTO to 0, and send function 10B (for a 501 printer) or 30B (for a 512 printer) to theprinter via subroutine DO. This cancels the effect of any previous functions that set automatic page skipping. However, it also cancels the effect of any previous selection of 8 lines/inch spacing on a 512 printer. So if the printer is a 512, and FL.8LL is 1, we send function 10B to the printer to restore 8-line spacing. Whether this is done or not, we arrive at SPCV.

If FORMAT does not contain Q, we go to SPCA. If it contains R, this calls for automatic skipping from bottom to top of page. We set FL.AUTO to 1, which will cause the appropriate function to be given to the printer before every line, and then go to SPCV.

If FORMAT does not contain Q or R, we go to SPCB. If it contains S or T, this calls for 6-line or 8-line spacing; we put 0 or 1 in D.Z1 and go to SPCP. At SPCP, if the printer is a 501, the switch will have been set to send us straight to LPRW, as there is no choice of spacing, and the line is to be simply ignored. The program behaves as if the line had not existed; with the single exception that if the line beginning with S or T is more than 155 bytes long, the remainder after 155 will be printed as though the

preceding 155 bytes had not been there and the 156th byte had 00B as its first character. For a 512 printer, however, we set FL.8LL to the number in D.Z1, 0 or 1. Then send the function 11B-(D.Z1), i.e., 11B for S or 10B for T, to the printer via subroutine DO. This will select 6-line or 8-line spacing.

DO is called by branching to SPCN; thereafter, we pick up the character in NCHAR, which is either the first character of the next line or zero, and call subroutine DUM to do the equivalent of a pre-print skip on the next line if necessary. When possible, a pre-print skip on the next line is handled as a post-print skip on this line; but this line is not printed at all, so DUM has to be called. Then we return to LPRW.

We come to SPCV afterissuing the necessary functions for a change to or from automatic page skipping. The line that began with Q or R is not to be printed, but there should be a skip to top of page before printing the next line. However, if the next line begins with ≠1≠ the format character that would have that effect, there is no point in having two page skips. So we see if NCHAR contains ≠1≠; if so, we go out to LPRW; if not, put 4 in the A register, the function code for page skipping, and go to SPCN as in the preceding paragraph.

If FORMAT does not contain Q, R, S, or T, we check for A or B, and if neither, go to SPCC. A and B call for skipping to top or bottom of page after printing this line. If this line is about to be printed on the line next before the top or bottom of page, and we allow the natural post-print skip to occur, then the explicit skip to top or bottom thereafter will be a skip of exactly a whole page; probably not what was intended. So we now send function 6 to the printer via subroutine DO, to suppress the natural post-print skip; then call subroutine PRINT to print the line; then put 4 or 3 in the A register, according as top or bottom is to be skipped to, and go to SPCN, where we send the function to the printer; obey the first character of the next line (NCHAR) if it calls for a pre-print skip; and go to LPRW.

If we have excluded A, B, Q, R, S, and T, we come to SPCC and try for a letter between C and L, which calls for a post-print skip to some channel. If not go to SPCE. If so, call subroutine POST with a corresponding number between 0 and 9 in the A register; the subroutine will send the corresponding post-print function code to the printer unless the format character was I through L and the printer is a 501, for which these characters have no meaning. If a post-print skip function is sent to a 501, subroutine POST will also set FL.POST to 1, so that the

clear function will be sent to the printer after the line is actually printed. After calling POST, we call subroutine PRINT to print the current line; then skip around the DO call at SPCN to pick up what may be the first character of the next line, in NCHAR, and send it to subroutine DUM to obey it if it calls for a pre-print skip on the next line. Since we had a post-print skip on this line, we could not handle a possible pre-print skip on the next line as a post-print skip on this one, so calling DUM was the only way to deal with the possibility. Then go back to LPRW.

If we have excluded A through L and Q through T, we come to SPCE and look for 1, 2, 0, or -. If none of them, go on to SPCG; to which, as noted before, we go if the first character of this line is 00B or any character not recognized as a special format character. If 1, 2, 0, or - is found, we put the corresponding function code 4, 3, 1, or 2 in the A register and pass it to the printer via subroutine DO at SPCJ, for a pre-print skip to top of page, skip to bottom of page, single extra space, or double extra space. Then arrive at SPCG.

At SPCG, we have finished with the format character of this line, but have not yet printed it, and are still in a position to obey the first character of the next line, if possible, as a post-print function on this line. If NCHAR contains +, 1, or 2, we go to SPCGA and send function 6 to the printer, to suppress the natural post-print skip on this line. For +, the reason is that it calls for the next line to be printed on top of this one, so suppressing the natural skip on this one will accomplish it. Note that this is the only place in the program where the format character + is recognized. For 1 or 2, we suppress the skip to avoid the possibility that the present line is about to be printed on the line just before the top or bottom of page, and that the skip that the 1 or 2 will produce when the next line is processed will be a skip of exactly a whole page, which presumably was not intended. After the suppress function in any of these three cases, we go to SPCGE to print this line.

If NCHAR does not contain +, 1, or 2, we pass the character to subroutine EV, to see if it is between 3 and 9, or X and Z, and calls for a pre-print skip on the next line. If not, the exit from EV is with the A register negative, and we branch to SPCGE. If so, the exit is with a corresponding number between 0 and 9 in the A register, and we call subroutine POST to issue the post-print function to the printer if possible and set FL.POST if necessary. Then, at SPCGE, we call subroutine PRINT to print this line and then return to LPRW.

Subroutines

XFR

This is called from two points in the main routine of 2LP to bring the next word, if any, from the circular buffer and put it in the next free space in the print line image area; namely in BUF+n through BUF+n+4, where n is the byte count in WC. If a word is moved, 5 is added to the content of WC. First, if the IN and OUT pointers in the PP are equal, we exit from XFR with 0 in the A register, showing that nothing has been moved and the buffer is empty. Otherwise we copy the word D.OUT points to into BUF+n ff., and add 5 to the value of n in WC. Then add 1 to the pointer in D.OUT and D.OUT+1; but if this makes it equal to D.LIMIT and D.LIMIT+1, reset it equal to D.FIRST and D.FIRST+1. Then exit from XFR with non-zero in the A register, indicating that a word has been moved.

Entry Information

Only the PP buffer pointers and the line image length in WC.

Exit Information

0 in the A register if the buffer was already empty; otherwise the A register contains non-zero, D.OUT and D.OUT+1 have been updated, and 5 has been added to the line image length in WC.

Other Subroutines Called

None.

Registers Destroyed

None.

WAIT

This is called to make the program wait until the operator types ▽n.GO.▽ First we set bit 12 of the word at RA+0 to 1. Then, at PSE3, call subroutines RES and REL to reserve and release the channel, 6681 and printer. This seems rather pointless, but subroutine REL includes a pause for storage relocation, and the possibility of dropping the control point. Then we check bit 12 of RA+0; if still 1, wait one millisecond and return to PSE3 to repeat the cycle. If the bit has become 0, the operator must have typed ▽n.GO.▽ We zero the B-display word in the control point, which contains ▽GO▽, and then exit.

Entry Exit Information

None, except that D.RA and D.FL have been updated if necessary during the call to REL.

Other Subroutines Called

RES, REL.

Registers Destroyed

Only D.T0 through D.T4 by WAIT directly.

DO

This is called whenever a special function is to be sent to the printer. The function code is in the A register on entry, and is saved. Subroutine RES is called to reserve the channel, select the 6681, and connect the printer. Then the function code is put back into the A register and sent to subroutine FCN, which sends it to the printer. Then subroutine REL is called to release the channel, 6681, and printer, and we exit from DO.

Entry Information

The function code in the A register.

Exit Information

None.

Other Subroutines Called

RES, FCN, REL.

Registers Destroyed

None by DO directly.

PRINT

This is called to print the current line, which begins at BUF, and whose length in bytes is stored at WC. We call subroutine LIST with address BUF in bits 0-11 of the A register, and address WC in bits 12-17. This makes LIST send the series of bytes so defined fφr printing. On return, the status byte of the printer has been stored in STAT; if bits 2 and 10 of this are both zero, there is no printer error and we exit from PRINT. Otherwise, we know the printer must be a 512, as these bits are not set by

the 501; so as the trouble may be in the print image memory of the 512, we call subroutine LOAD to re-load it. Then send function code 4 to the printer via subroutine DO to cause a page skip; call subroutine MSG to set up message LP uu PRINT ERROR; subroutine R.DFM to send it to the dayfile and put it on the B-display; and subroutine WAIT to wait until the operator types "n.GO." Then reset D.OUT and D.OUT+1 from the FET OUT pointer, and return to LPDA to try the same line again.

Entry Information

WC holds the length of the line in bytes.

Exit Information

None.

Other Subroutines Called

LIST, LOAD, DO, MSG, R.DFM, WAIT

Registers Destroyed

Only D.T0 through D.T4 by PRINT directly.

LOAD

This is called initially to load, or subsequently to re-load, the print image memory of a 512 printer with 288 characters taken from the right halves of bytes IMAGE through IMAGE+287. The characters are in display code, and must be sent to the printer in BCD; the left half of each byte contains display code 0, so that when sent to the printer it will be the corresponding code, 00B. These 288 characters correspond to the 288 characters in the same order on the train of the 512.

Subroutine LOAD also sends function 5 (set auto-page-skip status) to the printer if FL.AUTO = 1. This is necessary on the 512 initially or after a print error, and it is necessary initially on a 501 printer. So the only time LOAD is called for a 501 printer is during initialization, and in this case the jump at LOADS is suppressed so that everything between there and LOADA is skipped.

First (for a 512) we call subroutine DO to send function code 12B to the printer; this conditions the printer to copy the next print line sent to it into its print image rather than printing it. Next, we are about to call subroutine LIST to send the "line" of characters to the printer.

However, LIST will convert the characters from display code to BCD using table ALPH. Normally, the table converts both display code 55B and display code 00B into 60B, the BCD blank. But the printer train contains some per cent signs, which because of a limitation in SCOPE can never be printed. We must send to the corresponding positions of the image memory some BCD code that will never be sent to the printer during actual printing. The BCD code that is never used in printing is 16B; so we store 1600B in the position of table ALPH that corresponds to display code 00B; thus, temporarily, letting display code 00B take care of the unprintable per cent sign. Then we put the length of the image line, 440B or 288, into D.T7, and call subroutine LIST with address IMAGE in bits 0-11 of the A register and address D.T7 in bits 12-17. LIST sends the line to the printer. Then we restore table ALPH so that display code 00B will once more be turned into BCD 60B, the blank character.

This loading or reloading may have upset a previous selection of 8 lines/inch spacing on the printer; so we send function 30B to the printer through subroutine DO to clear 8-line and auto-page-skip selections; then send code 10B or 11B, to select 8-line or 6-line spacing according to flag FL.8LL; then send code 5, to select auto page skipping, if flag FL.AUTO is 1. This last selection, beginning at LOADA, is all that is done (if necessary) when LOAD is called during initialization for a 501 printer.

Entry and Exit Information

None, but the printer train image must have been set up or preserved in IMAGE through IMAGE+287.

Other Subroutines Called

DO, LIST

Registers Destroyed

Only D.T7 by LOAD directly.

LIST

This is called by subroutine PRINT to print the current line; by subroutine LOAD to send the printer train image to the printer image memory; and by subroutine DUM to print a line of two blanks after a post-print skip has been selected on the printer; this last is how a preprint skip for the next line is simulated when it cannot be handled as a post-print skip on the preceding real line.

On entry, bits 0-11 of the A register contain the address of the beginning of the line, which is stored at LISTA+1, and bits 12-17 contain the address of a direct cell containing the number of bytes in the line. This number is fetched and saved in D.T7.

At LISTD we call subroutine RES to reserve the channel, select the 6681, and connect the printer. Then call subroutine STS, which puts the status byte of the printer in the A register. If bit 1 is 0, the printer is ready and not busy, and we jump to LISTC. Otherwise, call subroutine REL to release everything, and return to LISTD to try again. At LISTC, we zero the B-display message word in the control point, which might have had a hardware trouble message put in it. Then function the printer for output of data, activate the channel, and zero D.Z1, the byte counter for the print line.

At LISTA, we pick up the byte D.Z1 points to, convert it from display code to BCD, send it to the printer, and wait till the channel is empty. Then add 1 to the count in D.Z1, and if it does not yet equal the line length in D.T7, return to LISTA.

When the whole line has been transmitted, we arrive at PRT7, disconnect the channel, call subroutine REL to release the channel, 6681, and printer, and set ALARM to contain the current time plus AL.LP milliseconds, modulo 10000B milliseconds. This is to prevent us from starting to process the next line until at least AL.LP milliseconds have passed, on the assumption that until then it is worse than useless to keep testing the printer status.

The switch at LISTS will make us exit from LIST if the printer is a 512. But for a 501, we test FL.POST; if this is 0, exit from LIST. If it is 1, we sent a post-print skip code to the printer before printing the line, and must now clear it; so we reset FL.POST to 0 and send the clear code, 10B, to the printer via subroutine DO. But if the printer is supposed to be in auto-page-skip status, this 10B function will have cleared that as well; so now, if FL.AUTO is 1, we renew auto-page-skip status by sending function 5 to theprinter. Then exit from LIST.

Entry Information

The A register contains the starting address of the line to be printed in bits 0-11, and bits 12-17 contain the address of a direct cell that contains the length of the line in bytes.

Exit Information

None, except that ALARM has been updated.

Other Subroutines Called

RES, STS, REL, DO

Registers Destroyed

D.Z1, D.Z2, D.T0 through D.T4, D.T7.

EV

This is called with either the first character of the current line in the A register, or with the content of NCHAR, which is either the first character of the next line or zero if that character is unknown. If the character is display code 3 through 9, we exit from EV with 0 through 6 in the A register. If it is display code X through Z, we exit with 7 through 9 in the A register. If it is anything else, we exit with a negative number in the A register.

Thus we exit with a negative number in the A register if the entry character does not call for a pre-print skip on the line in question, or with a positive number specifying the channel to which a pre-print skip is demanded. To exit numbers 0 through 9, the corresponding channels for the 512 printer are 6, 5, 4, 3, 2, 11, 7, 8, 9, and 10. To exit numbers 0 through 5, the corresponding channels for the 501 printer are 6, 5, 4, 3, 2, and 1. For exit numbers 6 through 9, no skip is done for the 501.

Entry Information

The A register contains either the first character of a print line, or zero indicating that the first character of the next line is unknown.

Exit Information

The A register is negative if the entry character did not call for a pre-print skip; otherwise it contains a number between 0 and 9, specifying the format channel for a pre-print skip.

Other Subroutines Called

None.

Registers Destroyed

None.

POST

This is called with a number between 0 and 9 (00B and 11B) in the A register, specifying a printer format channel to which a post-print skip is to be sent to the printer. This number may have been constructed by subroutine EV from a character calling for a pre-print skip (which is always simulated by a post-print skip), or by the main routine at SPCC from the first character of the present line if it calls for a post-print skip.

The switch at POSTS sends us to POSTZ if the printer is a 512. If the entry number is 0 through 4, this calls for a skip to channel 6 through 2; we have subtracted 5 from the number, giving -5 through -1; we change the sign of this and add it to 31B, giving a function code 36B through 32B which is sent to the printer via subroutine DO. If the entry number is 5, we send function code 5+36B= 43B to the printer, giving a skip to channel 11. If the entry number is 6 through 9, we subtract 5 from it and then add 36B, giving a function code of 37B through 42B, calling for a skip to channel 7 through 10. After calling subroutine DO, we exit from POST with a negative number in the A register, showing that a skip function has been sent to the printer.

The switch at POSTS is a no-op if the printer is a 501. If the entry number is 6 through 9, we exit from POST with a positive A register, showing that no skip function was sent to the printer, as format characters 9, X through Z, and I through L have no meaning for it. Otherwise, we have subtracted 6 from the number, giving 777771B through 777776B. Forming the logical difference of this with 777767B gives 16B through 11B, the function code for a post-print skip to channel 6 through 1, and this is sent to the printer via subroutine DO. Then we set FL.POST to 1, indicating that immediately after the next time a line is printed, the clear skip select code 10B must be sent to the printer. Then exit from POST with a negative number in the A register, indicating that a skip function has been sent to the printer.

DUM

This is called before doing anything else for the current line if FL.N2NXT is 1, indicating that if its first character calls for a pre-print skip, it has not already been simulated; in this case it is entered with the first character of the current line in the A register. It is also called if the current line had a post-print skip format character, so that a pre-print skip on the next line has still to be simulated. In this case, DUM is

called with the content of NCHAR in the A register, which is either the first character of the next line or zero if that character is unknown. If zero, DUM will do nothing, and FL.N2NXT will be set 1 when the next line begins to be processed, so that DUM will be called once more.

When DUM is entered, subroutine EV is immediately called, and if the exit from EV is with a negative A register, the character did not call for a pre-print skip and we exit from DUM. Otherwise, we immediately enter POST without changing the key number EV left in the A register. POST will either issue the corresponding post-print skip function and exit with a negative A register, or do nothing and exit with a positive A register. If the exit from POST is with a positive A register, we exit from DUM. Otherwise, we call subroutine LIST in such a way as to "print" a line consisting of the single byte 0000B, i.e., two blanks. The post-print skip will take effect immediately after this empty print action, thus giving the effect of a pre-print skip immediately before the next real line to be printed, whether the current or the next one. Then we exit from DUM.

Entry Information

The A register contains the first character of the current line or the next line, or zero if we are dealing with the possible next line but its first character is unknown.

Exit Information

None.

Other Subroutines Called

EV, POST, LIST

Registers Destroyed

None by DUM directly.

STS

This is called when the channel, 6681, and printer have already been reserved, selected, and connected by subroutine RES, to put the printer status byte in STAT and in the A register. It does not exit until the printer status is ready, though it may be busy or not.

We function the printer for status, activate the channel, input the status byte, deactivate the channel, and store

the byte in STAT. If bit 0 is 1, the status is ready; we go to STS5 to check bit 11. If this is 1, it may indicate that the printer is reserved by another channel (however, this cannot happen under SCOPE unless the printer is somehow connected to more than one computer.) If the bit is 0, we return the status byte to its original position in the A register and exit from STS. If it is 1, but the printer is a 512, we do the same because the bit has a different meaning. But if the printer is a 501, switch STS6 lets us continue; we call subroutine MSG to put the message LP uu RESERVED on the B display; call subroutine REL to release everything, pause for storage relocation, and allow a control point drop; call subroutine RES to reserve, select, and connect once more, and return to STS1 to try the status again.

If bit 0 of the printerstatus is 0, we do not check bit 11, but call subroutine MSG to put the message LP uu NOT READY on the B-display; call subroutines REL and RES, and return to STS1 to try the status again.

Entry Information

None, but the channel, 6681, and printer have already been reserved, selected, and connected.

Exit Information

The status byte of the printer is in STAT and in the A register, and is known to be ready and not reserved by another channel.

Other Subroutines Called

MSG, REL, RES

Registers Destroyed

None by STS directly.

MSG

This is called to copy the variable part of a message into MSGA+3 ff. (MSGA through MSGA+2 contain 'LP' followed by the EST ordinal of the printer) and to copy the whole message beginning at MSGA into the B-display message area of the control point.

Entry Information

The address of the beginning of the variable part of the message is in the A register. The message is terminated by a zero byte.

Exit Information

None.

Other Subroutines Called

None.

Registers Destroyed

D.Z1, D.Z2

RES

This is called to reserve the channel, select the 6681, and connect the printer. First we find the current time in milliseconds, modulo 10000B, subtract it from the value in ALARM, and add 10000B to the result if negative. Now either: (1) ALARM was recently set to the then time plus AL.LP msec.; if not more than AL.LP msec. have elapsed since then, the result of the above operation will be less than AL.LP+1; i.e., the alarm has not struck, and we do not want to bother the system by asking about the printer uselessly, so we wait one millisecond and return to RESF to try again. (2) ALARM was recently set to the then time plus AL.LP msec.; if more than AL.LP msec. but less than 10000B msec. (4 sec.) have elapsed since then, the result of the above operation will be greater than AL.LP; i.e., the alarm has struck and we go ahead to RESG. However, if 10000B msec. have elapsed since the alarm was set, it will appear that the alarm has not yet struck; the same may happen the first time RES is called since 2LP was loaded, as we do not bother to initialize ALARM. However, this error means only an extra delay of AL.LP msec., which can be ignored when added to an occasional delay of 4 seconds.

At RESG, we set ALARM to the current time in milliseconds, modulo 10000B, so that if we do not succeed in printing a line immediately, which would set the alarm ahead by AL.LP msec., at least the next time RES is called (this should be in a very few milliseconds) there is no doubt that the alarm will appear to have struck and there will be unnecessary delay.

Now at RES1 we take the 1 to 4 channels named in theprinter EST entry and fill them into a request to monitor to reserve one of them for our PP. Then call R.RCH to issue the monitor request. On return, we pick up the number of the granted channel, and store it in CH. If it is the same as the channel specified in any 2LP instruction that

names a channel {RES4 is used arbitrarily for the comparison} we go to RES3; if not, we first call subroutine R.STB with a pointer to the list at RESA. The first address in the list is CH, showing where to find the new channel address; the remaining addresses are those of all the instruction bytes in 2LP that specify the printer channel. R.STB will alter all those instructions so that they specify the channel named in CH. The list at RESA terminates with a zero byte. Then go to RES3.

At RES3 we select the 6681, prepare to send an equipment connect code, send the equipment connect code, which is already formatted in bits 0-11 of the EST entry, wait till the channel is empty; then set the return address of subroutine CFR to RES91 and enter CFR at CFR2. The effect of this differs from a simple RJM CFR in avoiding the FAN instruction with which CFR begins. FAN would send a function code to the equipment and then check for acceptance; the connect code cannot be sent by a FAN, so we send it by an OAN in subroutine RES and then call CFR to check on acceptance. If the A register contains zero on return from CFR, the connect was accepted and we exit from RES. If not, CFR has already released the channel, 6681, and printer, and we return to RES1 to try again.

Entry and Exit Information

None, except what concerns ALARM, and the fact that the channel, 6681, and printer are not reserved, selected, and connected on entry, and are so on exit.

Other Subroutines Called

R.RCH, R.STB, CFR.

Registers Destroyed

D.T0 through D.T4

REL

This is called whenever a printer action has just been completed, or found to be impossible for the moment, to release the printer, 6681, and channel, to pause for storage relocation, and to drop the PP if the error flag is non-zero. The channel and 6681 must be reserved and selected on entry; the printer will normally be connected but in fact we may have failed to connect it. First we send function code 0 to the printer, which disconnects it if necessary; then deselect the 6681; and finally call R.DCH to release the channel. Finally, call R.PAUSE; update D.RA and D.FL; exit if the error flag byte in the control point is zero; otherwise drop the PP.

Entry and Exit Information

None, except that the channel, 6681, and printer are reserved, selected, and connect on entry, and all released on exit.

Other Subroutines Called

R.DCH, R.PAUSE, R.MTR

Registers Destroyed

None.

FCN

This is called, with a printer function code in the A register, whenever a function is to be sent to the printer after it has been connected, and apart from printing itself. On entry, we save the function code at FN. The channel, 6681, and printer have already been reserved, selected, and connected. We call subroutine STS to get the status of the printer; it will not exit until the status is 'ready'; but we must still check the 'busy' bit. We call STS until the 'busy' bit is 0. Then pick up the function code and call subroutine CFR, which will send it to the printer and check for acceptance. If the function is accepted, the exit from CFR is with zero in the A register; we then zero the first word of the B-display line in the Control point, in case a hardware trouble message had appeared there in the meantime, and then exit from FCN. If the exit from CFR was with non-zero in the A register, the function was not accepted, and the channel, 6681, and printer have already been released. So we call subroutine RES to reserve, select, and connect them again, and return to FCN1 to try once more to send the function. We do not call STS on the re-try, because we cannot exit from RES until the printer status is 'ready' in any case; the only other status we were checking after getting the status was 'busy', but the printer can hardly be busy if it is reserved to this control point and file.

Entry Information

The printer function code in the A register. The channel, 6681, and printer are already reserved, selected, and connected.

Exit Information

None. The function code has been accepted, and the channel, 6681, and printer are still reserved, selected, and connected

CFR

This is called to send to the printer any function code that can be sent by a FAN instruction, and to check for acceptance. On entry, the function code is in the A register, and the FAN instruction is immediately executed. After that, from CFR2 down, everything concerns checking for acceptance.

CFR is also entered from subroutine RES in such a way as to skip the FAN instruction, as RES has already transmitted the relevant function, but to go through all the checking for acceptance.

At CFR2 we function the 6681 to ask for its status, activate the channel, input the status byte, deactivate the channel, and zero all bits but 0, 1, and 2. If those three bits are all 0, the function was accepted and we exit with 0 in the A register.

If bit 0 is 1, we go to CFR7; call subroutine MSG to format the message LP uu XMSN PARITY ERROR; call subroutine R.DFM to put it on the dayfile and the B-display; wait one second; issue a channel clear function; go to CFR6. The one-second wait is an effort to let any data transmissions that may be happening on the same channel, involving other PP's, be completed rather than interrupted by the channel clear. No new transmission could be begun by another PP during the wait, since the channel is reserved by our PP.

If bit 0 of the 6681 status is 0 but bit 1 or bit 2 is 1, we call subroutine MSG to format the message LP uu REJECT; then copy it to the B display line of the control point and arrive at CFR6.

At CFR6 we call subroutine REL to release the channel, 6681, and printer, pause for storage relocation, and give the opportunity to drop the PP. Then exit from CFR with non-zero in the A register, indicating failure.

Entry Information

A printer function in the A register. The channel, 6681, and printer are reserved, selected, and connected, except that the printer may not have been successfully connected when CFR is called to check the acceptance of the connect code.

Exit Information

If the A register contains 0, the function has been accepted by the printer and the channel, 6681, and printer have not been released. If the A register contains non-zero, the function was not accepted and they have all been released.

Other Subroutines Called

MSG, REL, R.DFM

Registers Destroyed

None by CFR directly.

2PC On-Line Punch Driver

General

2PC is a PP program, always loaded at 2000B. It is the only program that actually drives a card punch unit. It is called

1. by CIO to write a local file that has been assigned to a punch unit,
2. by PBC to punch out an area of central memory,
3. by 1PO, which was requested by 1OT, to punch a file from the output stack. In cases 2 and 3, there is no FNT entry for the cards regarded as a file.

Function

2PC punches cards in three different formats:

1. Standard binary. Each card has xy05 octal in column 1, where xy is the word count; a checksum in column 2; xy words of information beginning in column 3 up to column 77 if xy has its maximum value of 17B; and a sequence number within its record in columns 78-80, the first card of a record being numbered 1, not 0. Only the last card of a record may have xy less than 17B.
2. 80-column binary. Each card contains 16 words of information in its 80 columns. The last card of a record may in fact contain fewer than 16 words, but as there is no word count, this merely means that columns at the end of the card, in which no information is being transferred, contain zero.
3. Coded (BCD). Each card contains up to 80 BCD characters, beginning in column 1. As words are being transferred from the buffer to the card image, the last byte of each word is checked for being zero, and if zero, no more information is transferred to the card; the next word will begin the next card. When eight words have been transferred to the card image without finding a zero byte at the end of word, the next word in the buffer will simply begin a new card unless it is a zero word that was put in the buffer in order to have a word ending in a zero byte. Then the terminating word is skipped, and the next word begins the next card.

Unfortunately, card punching is unique among outputs in that the checking of a card cannot be done until just after the next following card is punched. E.g., card 3 in a record is checked just after card 4 is punched. Now if card 3 turns out to be bad, the programs has to offset both card 3 and card 4, and then back up its pointers not one card, but two. To make it worse, 2PC has to be able to work with CIO; this means that the PP may be dropped in the middle of a record, and the next time CIO and

2PC are loaded, they may have to back up the pointers and repunch something that was punched the last time they were loaded. So it has to keep all the necessary information in the FET and FNT.

In fact, there are three different OUT pointers. Let us call them x, y, and z. At the beginning of a record, they are all equal. Then pointer x is advanced in transferring data from the buffer to a card image. Then the first card is punched. There is no question of checking, as there is no preceding card. So z is set equal to y, and then y is set equal to x. Now z points to the beginning of the last card punched, and x and y point to its end+1 in the buffer. Then words are transferred from the buffer to the card image for a second card, while pointer x is advanced. Then this card is punched, and the preceding card is checked. At this point y and x point to the beginning and end+1 of the card just punched, and z and y point to the beginning and end+1 of the preceding, just checked, card. If the check is bad, we must reset x and y equal to z. If good, we can set z equal to y, and then y equal to x, and repeat the cycle. If punching is interrupted in the middle of a record by a lack of data in the buffer, there will be doubt about whether the last card punched was good. Just before finding that there was insufficient data in the buffer for another card, we will have been using x pointer to move the data. Y will still point to the beginning of unused data in the buffer, and z to the beginning of the punched but unchecked card. Now we must not set the OUT pointer in the FET equal to y, as somebody might then read data into the buffer and destroy the information between z and y, which we may need if, when 2PC is later reloaded, the last card punched is found to have been bad. So we store z in the OUT pointer of the FET. But in order to give 2PC a y pointer the next time it is loaded, since it is from y that it will have to start moving data for a new card image before it can even check the last card, the number of words that were moved to the last card is stored in the FNT, in bits 18-23 of the second word. The next time 2PC is loaded, it will get the z pointer from the FET, then construct a y pointer using this number from the FNT, then set x=y and start building the next card image.

The z pointer, in fact, is the one maintained in the FET in central memory (if 2PC is called by 1PO or CIO) or in cells D.BA and D.BA+1 (if 2PC is called by PBC, which uses no FET.) The y pointer is kept in cells D.OUT and D.OUT+1, and the x pointer in PA and PA+1.

Entry Information

D.FNT through D.FNT+9 hold the second and third words of the FNT entry for the file of cards being punched (if 2PC is called by 1PO or PBC, there is no FNT entry really, but this is simulated). D.EST through D.EST+4 hold the EST entry for the punch unit. But D.EST+3, which would normally contain the display code letter "CP", will contain zero if 2PC was called by PBC; this is merely a self-identification made by PBC. If 2PC is called by PBC, D.OUT, D.OUT+1, D.IN, and D.IN+1 contain the starting and ending+1 addresses, relative to RA, of the CM area to be

punched out; and the information in the input register does not directly concern 2PC.

Otherwise, D.FIRST, D.FIRST+1, D.IN, D.IN+1, D.OUT, D.OUT+1, D.LIMIT, and D.LIMIT+1 contain the normal pointers from the FET of the file, and the input register contains, in bits 0-17, the address relative to RA of the first word of the FET.

If no card from the present record has been punched yet, bits 0-23 of the second word of the FNT entry, in D.FNT through D.FNT+4, contain zero. Otherwise, bits 0-16 contain the number of cards already punched, bit 17 is a flag that does not appear ever to be used, and bits 18-23 contain the number of words of information that were taken from the buffer for the last card punched. D.OUT and D.OUT+1 contain the z pointer, as explained in "FUNCTION" above; and this number from the FNT enables us to construct the y pointer.

Exit Information

D.FNT through D.FNT+9 have been updated. Note that the sequence number of the last card punched has been stored in bits 0-16 of the word in D.FNT through D.FNT+4, and the number of words of information that were taken from the buffer for it has been stored in bits 18-23 of the same word. The OUT pointer in the FET (or cells D.BA and D.BA+1 if 2PC was called by PBC) has been set to point to the beginning of the last card punched, while D.OUT and D.OUT+1 have been set to its end+1 in the buffer. However, if 2PC has just punched an eor or eof card, bits 0-23 of the word in D.FNT through D.FNT+4 have been set to zero, and the IN and OUT pointers in the FET have been set equal to the FIRST pointer.

Other Programs Called

None.

Narrative

First subroutine PRS is called. As this is the only place it is called, we describe it here as part of the main routine.

PRS begins by formatting information from the EST entry into 6681 select and equipment connect codes. Then it inserts the equipment number in the message at MSGA. Then it sets the x pointer (PA) equal to the y pointer (D.OUT) and sets PM to 1. PM is 1 whenever the last card punched should be offset by giving a suitable function code right after punching the following card, and 0 otherwise. Now check bits 18-23 of the second word of the FNT entry. If these are zero, we have yet to punch the first card of the record, so the y pointer already equals the z pointer, and the x pointer should be equal to both, as we have now set it. Also, the last card punched was 7-8-9 eor card, and this should be offset. So

PM=1 is correct, and we exit from PRS. But if these bits in the FNT are not zero, they give the number of words that were taken from the buffer for the last card punched; and we know that the z pointer in the FET points to the beginning of that card; so did the y pointer when we entered PRS, and we have set the x pointer the same. Now we put the number in cell WC, and call subroutine XFR. This moves words from the buffer into the area beginning at BUF; we shall clear that area before we actually build up a card image in it; but what XFR has done usefully is to advance the x pointer, so that it now points to the end+1 of the last card punched. Now we set the y pointer (D.OUT) equal to the x pointer (PA) so that the x pointer is free to be used in moving data for the next card. Since we do not, so far as we know, want to offset the last card punched, we zero PM, and then exit from PRS.

After PRS, we check the mode in the FNT. If this is binary, we test the disposition code in the FNT, and branch to BIN or PAB according as it is 12B or 14B (normal binary or 80-column binary). If the mode in the FNT is BCD, we go to HOL.

We come to PCD4, just below where we choose a branch to BIN, PAB, or HOL, in only two cases - from PCD3+1 above, when the status in the FNT shows rewind, unload, or backspace status (this could scarcely happen) or from CHK5, after punching an eor or eof card, when not called by PBC. At PCD4 we merely set IN=OUT=FIRST in the FET before exiting from 2PC.

Before describing what happens at BIN, PAB, and HOL, let us describe subroutine CHK. This is entered with a number in the A-register: 15 when punching normal binary and 16 when punching 80-column binary cards, as this is the number of words we expect to move per card, or 1 when punching BCD cards, as we examine the words one by one for final zero bytes as terminators. This number is saved in WC, and then we look at the FIRST, IN, OUT, and LIMIT pointers in the PP (using the x pointer for OUT, not the y pointer) to see if there are that many words left in the buffer. If so, we exit from CHK, having merely put the word count in WC, but not having touched any pointers. If too few left, we put the number that are left into WC, as this may be the word count for the next card if it turns out to be the last data card of a record. (Note that for BCD punching, we always entered CHK with 1 in the A-register, so if there were fewer words than that left in the buffer, we must have just zeroed WC. This thought simplifies the routine beginning at HOL.) Now call subroutine CLR to zero the card image area (so that a less-than-full binary card at the end of its record will contain blank columns, not garbage, in its unused space. We need not do this for a full binary card, as the entire card image is filled with new things anyway.).

Now check the status in the FNT. If it is not eor or eof, exit from 2PC. We are not called on to do anything further until more has been read into the buffer. The z pointer is already in the FET, and the number necessary to reduce the y pointer, i.e. the number of words of information in the last punched card, as well as its sequence number in the record, have already been saved in the FNT.

But if the status is eor or eof, and WC, the number of remaining words in the buffer, is not zero, exit from CHK. This must mean exit back

to the routine at BIN or PAB, as for BCD punching, WC would have to be zero now. So we would be going back to that binary routine having put the initial 15 or 16 in WC, and then either confirmed that there were that many words left in the buffer, or having adjusted WC down to the number of available words. If the status is eor or eof, and WC=0, the buffer is empty and it is time for the eor or eof card to be punched. We have already zeroed the card image; now we put 0007B or 0017B in column 1. Then, for eor, we get the level number from the FNT, convert it to two octal BCD characters, and store them in columns 2 and 3. Then, at CHK3A, we call subroutine PCH to punch the card, and also to check the preceding card. If it found an error in the preceding card, PM will = 1, otherwise 0. If 1, it will have offset the preceding card, but the offsetting of the just-punched eor or eof card remains to be done. If 1, we lower the exit address in CHK by 3, and then exit. This means that the program immediately reloads the A-register with 16, 15, or 1, and re-enters CHK. But PCH (or rather subroutine CKC, always and only called by PCH) has also backed up the x pointer to equal the z pointer, so this time we come through CHK we will not set up the eor or eof card, but merely set WC = the number of words in the buffer remaining for the last card, and return from CHK to the routine that will set up that last card for punching once more. When that card is punched once more, CKC will see the 1 in PM, and will offset the previous card, i.e. the eor or eof card we punched in CHK, but found that its predecessor was bad.

But if the card before the eor or eof card was found to be good, so that PM = 0, we continue at CHK4. If it was an eor card, we go down to CHK6 to clear bits 0-23 of the second word of the FNT and then exit from 2PC, first setting IN=OUT=FIRST in the FET, at PCD4, if 2PC was not called by PBC. If it was an eof card, we first do a "4" function on the card punch, which causes the eof card to be pushed on out of the punch unit, and checked - we ignore the check - and a blank card is "punched" behind it. Then go to CHK6. The blank card behind an eof, or the eor card itself, will be offset because when 2PC is called for a new record, PM will initially be set to 1.

So we can say that apart from complications due to punch faults, CHK sets WC to the number that was in the A-register on entry, or to the number of words remaining in the buffer, whichever is smaller; and exits from 2PC if less than a full card is left and the status is not eor or eof; and if there are no words left in the buffer and the status is eor or eof, it punches the appropriate card and exits from 2PC.

Now for the punching of 80-column binary cards, beginning at PAB. We put 16, the number of words per full card, into the A-register and call subroutine CHK. If we return from CHK to the PAB sequence, we know that WC contains the number of words to be punched in the next card, beginning at what the x pointer points to. Se we put the address BUF in the A-register and call subroutine XFR, which will move the number of CM words given in WC from the buffer, beginning where the x pointer points, into PP memory beginning at BUF, i.e. the card image. XFR will update the x pointer accordingly. Next we call subroutine CCT, which merely adds 1 to the card count in bits 0-16 of the second word of the FNT entry, in

D.FNT through D.FNT+4. Finally, we call subroutine PCH to punch the card whose image is already in BUF through BUF+79, and to check the punching of the preceding card. If the preceding card was good, the z pointer will be advanced to equal the y pointer, and the y pointer will be advanced to equal the x pointer, and we then begin the cycle again at PAB. If the preceding card was bad, the x pointer will be backed up to the z pointer, and we recommence the cycle at PAB. In the latter case, the y pointer will not be moved, but this does not matter. Having backed up two cards, we will re-punch the first of them, but when we get into subroutine CKC we will find PM already = 1; so we will not check the preceding card but merely offset it, and will not set the z pointer equal to the y pointer. On exit from CKC on that occasion, the z pointer will still point to the beginning of the first of the two cards to be re-punched, and the x and y pointers will both point to the beginning of the second.

So much for the PAB cycle. The exit from it can take place only through subroutine CHK when the buffer is exhausted, with or without eor status.

The BIN cycle, for punching ordinary binary cards, differs from the PAB cycle only because the card format is more elaborate. First enter subroutine CHK with 15 in the A-register, as that is the number of words of information that fill a card. The logic of what subroutine CHK does is exactly the same as for the call on CHK in the PAB cycle. If we return from CHK into the BIN cycle, WC has been set to the number of words of information that are to be punched in the next card. Now we set the A-register to BUF+2, because information in a standard binary card begins in column 3, and we call subroutine XFR to move words from the buffer to the card image, updating the x pointer. Again, this is exactly the same in principle as for the PAB cycle. Now (here we differ from the PAB cycle) get the word count from WC, put it in the upper half of column 1, with a 5, for 7- and 9-punches, in the lower half. Then call subroutine CKS to form the checksum and store it in column 2 of the card image. Then call subroutine CCT, just as in the PAB cycle, to increase the card count in the FNT. Then zero column 78 (this is not in the PAB cycle) and copy the card count from the FNT into columns 79 and 80. Then, as in the PAB cycle, call subroutine PCH to punch and check, and re-commence the BIN cycle at BIN.

The HOL cycle, for BCD cards, is more complicated because we do not know in advance how many words we shall put in a card image, as the first word that ends in a zero byte will terminate the card. First call subroutine CLR, to zero the whole card image preparatory to moving in the images of from 1 to 8 BCD words. This subroutine also leaves cell D.Z1 containing the address BUF, and we shall use this setting in a moment as an index register as we fill in successive columns of the image beginning with column 1. Then zero D.Z4, in which we shall count how many words we took from the buffer in making up this card image (WC will not contain this count as it would for a binary card). Now we are at HOL1. We call subroutine CHKHOL to check the state of the buffer. If the status is end of record, and the buffer is already empty according to the x pointer, and D.Z4 does not contain zero, the record is exhausted but the card image

is not empty, and CHKHOL branches to HOL5 to punch out the card. If the record is exhausted but D.Z4 contains zero, the card image is still empty, so CHKHOL puts 1 in the A-register and calls the subroutine CHK, which will handle the eor card in that case. If the record is not exhausted, either because the status is not eor, or because the buffer is not empty, CHKHOL puts 1 in the A-register and calls subroutine CHK, which will simply leave 1 in cell WC if the buffer is not empty, or exit from 2PC if it is empty. Then we set the A-register to contain the address D.T0, and call subroutine XFER, which moves into D.T0 through D.T4 the word in the buffer to which the x pointer points, and advances the x pointer by 1. Then we add 1 to the word counter in D.Z4. Next we convert the 10 characters of the word into ten bytes, taken from table HOLA, which we store in 10 cells beginning at the one to which D.Z1 points - initially BUF for column 1. Table HOLA gives, for every display code character the corresponding pattern of holes in BCD card column. Now D.Z1 points to the next free cell in the card image. When D.Z1 comes to contain BUF+80, we have filled the card image. We do not know whether the last byte of the 8th word was zero, and this does not matter; yes or no, that word ends a card anyway. The only question is whether, according to the normal method of handling the suppression of trailing blanks, the line is terminated immediately after the 80th character by a whole word of zeroes; if so, we do not want that word to result in an extra blank card.

Before looking at the next word, we call subroutine CHKHOL to check the state of the buffer. If the status is end of record and the buffer is now empty according to the x pointer, the record is exhausted and CHKHOL branches to HOL5 to punch out this 80-character card. Otherwise, CHKHOL puts 1 in the A-register and calls subroutine CHK. If the buffer is not empty, CHK merely sets WC = 1 and returns. If the buffer is empty, CHK exits from 2PC; but this is all right because the status is not end of record, so it is not our last chance at this information. If we return from CHK an CHKHOL there is more available in the buffer, and we fetch the word the x pointer points to. If its first byte is not 0, we go to HOL5 to punch the card we have set up, knowing that the next word in the buffer can be allowed to begin another card. If that byte is zero, we assume the whole word is zero, and is merely a terminator to the card whose image we have just set up. So we call XFR with the address D.T0 in the A-register. As subroutine CHK has set WC = 1, XFR advances the x pointer by 1, skipping over the zero word, and copies the zero word into D.T0 through D.T4, where we are not interested in it any further. Then add 1 to D.Z4, because it represents, not the number of words in this card, but the number of words that have to be taken from the buffer for this card, which is in fact 9. Then go to HOL5 to punch this card.

Whenever we finish converting a word other than the 8th one, we get to HOL4. Here we test the last byte of the word from the buffer, now in D.T4. If the byte is not zero, we return to HOL1 to begin the next word from the buffer; if it is zero, we have moved enough words from the buffer, the x pointer is now correct, the number of words taken from the buffer is in D.Z4, and we go to HOL5.

We come to HOL5 when we are ready to punch out a BCD card. First call subroutine CCT, to advance the card counter in the FNT. Then set WC = D.Z4, the number of words taken from the buffer for this card, and call PCH to punch and check. Then return to HOL to continue the cycle. What PCH does with the number in WC is to store it in the FNT, in case this is the last card we punch before exiting from 2PC.

Subroutines

CCT

This adds 1 to the card count maintained in bits 0-16 of the second word of the FNT entry, in D.FNT through D.FNT+4. The only use made of this count is to insert it in column 79 and 80 of each normal binary card image. There is actually no need to call CCT in the PAB and HOL cycles.

Entry and Exit Information

Only the card count itself.

PCH and CKC

As the last action of subroutine PCH is always to call subroutine CKC, and as this is the only place CKC is called, we shall describe them together. When PCH is called, the card image has already been set up in BUF through BUF+79. We call subroutine RES to reserve the card reader. Then put 1 in the A-register, as a function code for selecting binary punching (since we have set up even a BCD card as a binary card image) and call subroutine FCN to issue the function to the punch unit. Then output the 80 words of the card image on the punch. Then call subroutine CKC.

Subroutine CKC begins by calling subroutine STS, which reads the status of the card punch and stores it at ST. Then we look at cell PM. Ordinarily, it contains 0, and we go to CKC1. However, it will contain 1 if this is the first card of the record that we have just punched, or if, when we punched the preceding card, we found that the second preceding card was badly punched. In either case,we send function code 3 through subroutine FCN to the punch, then release it with subroutine REL, zero PM, and go to CKC3. We offset the preceding card with this function code because if this is the first card of a record, the preceding one was an eor card and must be offset. Or if not, the second preceding card was bad and has already been offset, and the preceding card has to be offset as well; now the card we just punched above in PCH is a second attempt at the second preceding card. In both cases, by going this way to CKC3, we avoid checking the preceding card (i.e., we do not check the preceding eor card in one case, or in the other case, the second preceding card was bad, so the next preceding card had to be offset and abandoned, and there is no use in checking it).

Also, we do not advance the z pointer in the FET to equal the y pointer in D.OUT and D.OUT+1. In the case of the first card of a record, the z pointer already equals the y pointer. In the case of a re-punch, caused by bad punching of the second preceding card, we must continue to have the z pointer point to the beginning of the second preceding card in the buffer, because although we have just re-punched that card, we have not yet checked it.

But if PM=0 when we enter CKC, then after calling subroutines STS we go to CKC1. Here we check the error bit in the status word at ST; if this is 0 we continue by calling subroutine REL to release the punch; then we set the z pointer (in the FET unless we were called by PBC, or at D.BA if we were) equal to the y pointer. We now know that the preceding card was good, so we move the z pointer up to the beginning of the card just punched. Then go to CKC3.

If the error bit in ST is 1, we go to CKC4. Here we call subroutine FCN with 3 in the A-register, to offset the card just checked. Then call subroutine REL to release the card reader. Then call subroutine MSG to make up the message "CP xx COMPARE ERROR", and call subroutine R.DFM to put it out on the system dayfile but not the control point dayfile. Now set the x pointer equal to the z pointer, so that we shall start back two cards. Then go to CKC6, where we set PM=1, which will have the consequences described above when PCH and CKC are next called. Then reduce the card count in the FNT by 2, and exit from CKC and then from PCH.

On arriving at CKC3, which we do either if PM contained 1 on entry, or if it contained 0 and the check of the preceding card did not find an error, we set the y pointer equal to the x pointer, and store the content of WC, which is the number of words of buffer that went into the card just punched, and hence the number by which the z pointer must be advanced in order to equal the x and y pointers, in bits 18-23 of the second FNT word in D.FNT through D.FNT+4. Then exit from CKC and hence from PCH.

Entry Information

A card image in BUF through BUF+79. The x, y, and z OUT pointers, in PA and PA+1, D.OUT and D.OUT+1, and the FET respectively; except that the z pointer is in D.BA and D.BA+1 if 2PC was called by PBC. D.EST+3 contains 0 if 2PC was called by PBC, and not otherwise. The address of the first word of the FET (unless called by PBC) is in bits 0-17 of the input register.

PM contains 1 on entry if the card to be punched is the first of a record, or if the card two before the one now being punched was found bad.

Exit Information

If PM contained 1 on entry, it is left with zero, and the z pointer is not altered. If PM contained 0 on entry, then if the check of the preceding card is good, PM is left 0, and the z pointer is set equal to the y pointer in either of these two cases, the y pointer is set equal to the x pointer, and the number of words of buffer that went into the card just punched is

stored in bits 18-23 of the FNT second word in D.FNT through D.FNT+4. If PM contained 0 on entry, but the check of the preceding card was bad, PM is left containing 1, the x pointer is set back to equal the z pointer, and the card count in bits 0-16 of the FNT second word in D.FNT through D.FNT+4 is reduced by 2.

Subroutines Called

RES, FCN, STS, REL, MSG, R.DFM.

Registers Destroyed

D.TO through D.T4; also those destroyed by the above subroutines.

CHK

This is entered with the number of words wanted in the A-register: 16 in the PAB cycle for 80-column binary cards, 15 in the BIN cycle for normal binary cards, and 1 in the HOL cycle for each word to be fetched from the buffer. This number is stored in WC, which will tell subroutine XFR how many words to fetch, and then is compared with the number of words still in the buffer, using the x pointer as OUT pointer. If there are at least the wanted number of words in the buffer, we exit from CHK, which has done nothing but set WC to the number of words to be fetched.

If there are not enough words in the buffer, WC is set to contain the number of words that remain. Then subroutine CLR is called to clear the card image, so that an incompletely filled card will not have garbage in its unused columns. Then if the FNT shows the status is not eor or eof, we exit from 2PC as there is nothing further to be done.

If the status is eor or eof, we are at CHK2. Now if WC contains 0, showing that the buffer is empty, we proceed to CHK3 to punch the eor or eof card. Otherwise, exit from CHK, which has done nothing but set WC to some number less than the wanted 16 or 15, so that the PAB or BIN cycle will punch the remaining words in the buffer in a short last data card. At CHK3, set up the image of an eor of eof card, getting the eor level from the FNT if necessary. At CHK3A, call subroutine PCH (and CKC) to punch the card and check its predecessor. If now PM contains zero, the predecessor was good, and we go to CHK4. If PM contains non-zero, the predecessor was bad, and we shall have to re-punch it and the eor or eof card. We leave non-zero in PM, and return to the instruction before the return jump that got us into CHK. In effect, this means we load 15, 16, or 1 in the A-register and come back into CHK.

Now, however, the x pointer has been set back, in subroutine CKC, to equal the z pointer, which points to the beginning of the card that preceded the eor or eof card. On this second trip through CHK, we will merely set WC to the number of words for the last card of the record, or to 1 if we are punching BCD.

At CHK4, we test and branch straight to CHK6 if the card was eor. If it was EOF, we call subroutine RES to reserve the card punch, then call subroutine FCN to issue a 4 function code to it, then call subroutine REL to release it. The 4 function has the effect of punching a blank card after the eof, and checking the eof card, although we shall make no use of the check. The eof card is not offset. The card following it will not emerge from the card reader until the first card of a new record and file is punched, and at that time it will be offset.

At CHK6 we clear the card count and the words-in-last-card count in bits 0-23 of the second FNT entry word, at D.FNT through D.FNT+4. This means that the next time 2PC is called for this file, it will know it has a new record to deal with, and should offset preceding card without checking it. Then, if D.EST shows 2PC was called by PBC, we exit from 2PC immediately; otherwise, we set IN=OUT=FIRST in the FET (at PCD4) before exiting from 2PC.

Entry Information

The number of words wanted in the A-register; the IN pointer in D.IN and D.IN+1; the z-OUT pointer at PA and PA+1; the FIRST and LIMIT pointers at D.FIRST, D.FIRST+1, D.LIMIT, and D.LIMIT+1; the second and third words of the FNT entry in D.FNT through D.FNT+9; zero in D.EST+3 if 2PC was called by PBC, and not otherwise.

Exit Information

WC contains the smaller of the number that was in the A-register on entry and the number of words remaining in the buffer, using the z-pointer for OUT.

Subroutines Called

CLR, PCH (involving CKC), RES, FCN, REL.

Registers Destroyed

D.Z1, as well as those destroyed by the above subroutines.

CLR

This merely zeroes cells BUF through BUF+79, i.e. the whole output card image. It leaves D.Z1 containing the address BUF, and this is used as an index immediately after CLR has been called, in the HOL cycle and in subroutine CHK.

Entry Information

None.

Exit Information

D.Z1 contains the address of the beginning of the card image.

Subroutines Called

None.

Registers Destroyed

None.

CKS

This is called only in the BIN loop, to find the checksum of a normal binary card and insert it in column 2 of the card image, after all the information has been stored in columns 3 through 77.

It does this by adding each column to a cumulative total originally 0, but setting bits 12-17 of the accumulator to 77B before each addition, so that a carry out of bit 11 results in an addition of 1 to bit 0.

Finally, this total is subtracted from zero, and bits 0-11 of the result are stored in column 2 of the card image. The result is that if columns 2 through 77 are added up, with bits 12-17 of the accumulator set to 77B for end-around carry before each addition, the total should be zero.

Entry Information

Columns 3 through 77 of the card image.

Exit Information

Column 2 of the card image.

Subroutines Called

None.

Registers Destroyed

D.Z1, D.Z2.

XFR

This takes n central memory words from the circular buffer and stores them beginning at p in the PP memory, where n is the number in cell WC and p is the address in the A-register on entry to the subroutine. The z OUT pointer, in PA and PA+1, is used and updated. The IN pointer is not checked, as subroutine XFR is never called unless subroutine CHK has already confirmed that there are enough words remaining in the buffer.

Entry Information

The number of CM words to be moved, in cell WC; the z OUT pointer at PA and PA+1; the FIRST and LIMIT pointers.

Exit Information

The z OUT pointer has been updated.

Subroutines Called

None.

Registers Destroyed

D.Z5.

STS

This reads the status of the card punch, which must have already been reserved and connected. The status word is stored at location ST, and is also in the A-register on exit from STS. If bits 1 and 6 are 0 (busy bit and feed failure bit), and bit 0 is 1 (ready bit), we merely exit from STS. Otherwise, we call subroutines REL and RES to release and reserve the card punch, and go through STS again. If the ready bit or feed failure bit is wrong, we also display the message "CP xx NOT READY" on the B-display.

Entry Information

None.

Exit Information

The status word read from the punch, in the A-register and in location ST.

Subroutines Called

REL, RES, MSG.

Registers Destroyed

None, except those destroyed by the above subroutines if they are called.

MSG

This copies a message, whose starting address is in the A-register on entry, to MSGA+3ff. It stops with the first zero byte in the message. Then it copies MSGA through MSGA+14, i.e. "CP NN" where NN is the number of the card punch, followed by the message, into control point +35B, +36B, 37B.

Entry Information

The starting address of the message in the A-register.

Exit Information

None.

Subroutines Called

None.

Registers Destroyed

D.Z1, D.Z2.

RES

This reserves and connects the card punch. First it takes the numbers of the possible channels from D.EST+2 and D.EST+1, and calls subroutine R.RCH to reserve one of them. Then it stores the number of the obtained channel at location CH, and compares it with the number of the channel last used, if any. If they do not match, subroutine R.STB is called to insert the new channel number in all the instructions pointed to by table RESA. Then the 6681 is selected, and the punch is connected. Subroutine CFR is called to issue the connect code and check its acceptance. If on return the A-register contains zero, all is well and we exit RES. Otherwise, go through RES again.

Entry Information

The EST entry for the punch, in D.EST through D.EST+4. as modified by subroutine PRS.

Exit Information

The number of the reserved channel, in cell CH.

Subroutines Called

CFR, R.RCH, R.STB.

Registers Destroyed

D.Z1, D.Z2, D.T0 to D.T4.

REL

This releases the punch, the 6681, and the channel, and pauses for storage relocation. If the error flag is set, it drops the PP.

Entry Information

The EST entry for the punch, in D.EST through D.EST+4, as modified by subroutine PRS. The channel number in CH.

Exit Information

None.

Subroutines Called

R.DCH, R.PAUSE, R.PROCES

Registers Destroyed

D.T0 through D.T4.

FCN

This issues the function code that is in the A-register on entry, to the punch unit, which must already have been reserved and connected. The function code is saved in location FN. Then subroutine STS is repeatedly called until the status shows bit 1, the busy bit, is zero. Then we put the function code in the A-register again, and call subroutine CFR to send it to the punch. On return from CFR, if the A-register contains 0, all is well; we zero the message word at control point+35B and exit from FCN. If the A-register does not contain zero, subroutine CFR was not able to issue the code, and has already called usbroutine REL to release the punch. So we, in FCN, call RES to reserve and connect it again, then go back to pick up the function code and call CFR once more. This will be repeated until the code is successfully given to the punch.

Entry Information

The function code in the A-register.

Exit Information

The message word at control point+35B has been zeroed.

Subroutines Called

STS, CRF, RES.

Registers Destroyed

Only those destroyed by the above subroutines if called.

CFR

This is the subroutine that actually transmits function codes to the card punch. The function code is in the A-register on entry; it is immediately sent to the punch, and then the 6681 status is read. If bits 0, 1, and 2 of the status word are 0, all is well and we exit from CFR with 0 in the A-register. If bit 0 is 1, we go to CFR7; send the message "CP xx XMSN PARITY ERROR" to the system dayfile but not the control point dayfile, wait one second, clear the channel, call subroutine REL to release the card punch and the channel, and exit from CFR with 1 in the A-register. If bit 0 is 0, but one or both of bits 1 and 2 are 1, we merely put the message "CP xx REJECT" on the B-display, and then call REL to release the punch and the channel, and then exit from CFR with 1 in the A-register.

Entry Information

The function code for the punch is in the A-register.

Exit Information

0 in the A-register if the function code has been accepted by the punch; 1 in the A-register if not, and the punch has been released.

Subroutines Called

MSG, REL, R.DFM

Registers Destroyed

Only those destroyed by the above subroutines if they are called.

Messages from 2PC

CP XX COMPARE ERROR. This goes to the B-display and the system dayfile whenever a card is checked and found bad. (subroutine CKC)

CP XX NOT READY. This goes to the B-display whenever bit 0 of the punch status word is found to be 0, or bit 6 is found to be 1. (subroutine STS)

CP XX REJECT. This goes to the B-display whenever bit 0 of the 6681 status word is found to be 0, but bit 1 or bit 2 is 1. (subroutine CFR)

CP XX XMSN PARITY ERROR. This goes to the B-display and the system dayfile whenever bit 0 of the 6681 status word is found to be 1. (subroutine CFR)

No operator action is called for by any of these messages:

Index to Location Symbols in 2PC Flowcharts

Name	Page	Name	Page
BIN	1	MSG $	8
CCT $	3	PAB	1
CFR $	9	PCD	1
CFR2	9	PCD3	1
CFR6	9	PCD4	1,6
CFR7	9	PCH $	3
CFR9	9	PRS $	5
CHK	6	REL $	8
CHKHOL $	10	RES $	8
CHKHOLA	10	RES3	8
CHK1	6	STS $	7
CHK2	6	STS5	7
CHK3	6	STS6	7
CHK3A	6	XFR $	7
CHK4	6	XFR1	7
CHK6	6	XFR3	7
CKC $	4		
CKC1	4		
CKC3	4		
CKC4	5		
CKC5	5		
CKC6	5		
CKS $	3		
CLR $	3		
FCN $	9		
FCN1	9		
FCN2	9		
HOL	2		
HOL1	2		
HOL2	2		
HOL3	2		
HOL4	2		
HOL5	2		

$ indicates a subroutine entry.

2RC On-Line Card Read Driver

General

2RC is a PP program, in the form of a subroutine, always loaded at 2000B. It is called as an overlay by 1LJ at the "READ" control point, to read cards that will become job files, and by CIO at a job control point to read a local file that has been assigned to a card reader.

Function

2RC reads every card in the binary mode. Neglecting for the moment the possibility of 80-column status, it transmits information from the card image as follows:

1. If column 1 does not contain 7 and 9 punches, the 80 columns are translated into 40 bytes of display code characters on the assumption that it is hollerith-punched. Any column containing an invalid hollerith code is treated as blank. What is moved into the buffer is the information from the first byte up to and including the last byte that is not two blanks, followed by from 1 to 5 extra bytes, enough to make a multiple of 5 bytes (i.e. an integral number of CM words) of which the last 1 to 5 bytes are zero.

2. If column 1 contains 0017B, the card contains no information but marks an end of file.

3. If column 1 contains 0007, the card contains no information but marks an end of record. Columns 2-3 are then read as if punched in hollerith. If column 3 contains blank, and column 2 a number from 0 to 7, this number is the eor level number. If columns 2 and 3 contain an octal number between 00B and 17B, this is the eor level number. Otherwise the level number is taken as 0.

4. If column 1 contains xy05B, where xy is an octal number between 01B and 17, the card is a normal binary one, and contains xy CM words of information punched in columns 3 through 2+5*xy. Then column 2 must contain the correct checksum for the card; otherwise the job will be aborted. (If 2RC was called by 1LJ, the job must be "pre-aborted" by putting a signal in the FNT entry for the file, which 1BJ will translate to an immediate abort when the job is eventually brought to a control point.) Columns 79 and 80 should contain the correct binary serial number for the card within its record (the first card is number 1, not number 0), otherwise there will be an error message.

5. If column 1 contains xy45B, the case is the same as in (4) above, except that column 2 will not be tested to see if it contains the proper checksum.

6. If column 1 contains anything else, the card is unrecognizable, and the job is aborted (see (4) above).

This is complicated, however, by the possibility of 80-column status, which allows a user to read cards as containing 16 words of binary information each, without regard to their format. The first time, within a record, a card with 7777B in column 1 and in one other column, and 0000B in all other columns, is read this card contains no information, but puts the file into 80-column status. Until the next card identical with that one is read, all cards, no matter what their format, are read as 16 words of binary information a piece. Thus no end-of-record is possible while in 80-column status. The only exception is that a card with 0017 in column 1, and 0000B in all other columns, will mark an end-of-file even while in 80-column status, and will terminate 80-column status. This provision is made so that a normal end-of-file will be an absolute separator between jobs in the card reader, even if one job misuses 80-column status.

2RC exits back to 1LJ or CIO when it has processed an end of record or end of file card, or when it finds fewer than 17 vacant words in the buffer, so that there might not be room to store another card.

Entry and Exit Information

The second and third words of the FNT entry for the file are maintained in D.FNT through D.FNT+9. The EST entry for the card reader is in D.EST through D.EST+4 on entry to 2RC, and is left unchanged. The FIRST, IN, OUT, and LIMIT pointers are in D.FIRST, D.FIRST+1, D.IN, D.IN+1, D.OUT, D.OUT+1, D.LIMIT, and D.LIMIT+1 on entry to 2RC. The IN pointer is updated by 2RC, and is written back to the FET as information is transferred to the buffer.

Cell BS, at D.BA+4,contains the status of the file before the current call to 2RC, and 2RC does not touch it. It is set by overlay 2BP, which is always called by CIO or 1LJ just before 2RC is called.

The second word of the FNT entry for a card file, in D.FNT through D.FNT+4, has a special format:

```
VFD 6/a,11/b,7/c,12/d,6/0,1/e,17/f
```

a is the equipment code, 60B for a card reader.

b is the record count within the file, 0 for the first record.

c is the record type - 0 for binary, 1 for hollerith, between 2 and 80 for 80-column status, and 177B immediately after 80-column status is terminated, unless these record type numbers, see below under "Error Messages".

d is the EST ordinal of the card reader.

e is the end-of-job flag, set when and end-of-file card is read without and end-of-record card just before it. 2RC then processes as if for end-of-record; but the next time 2RC is called it recognizes the end-of-job flag and processes as if for end of file.

f is the card count within a record, which is turned from 0 to 1 on reading the first card of the record, and increased by 1 for each card thereafter. A normal binary card will have its serial number in columns 79-80 compared with this, and a discrepancy will give an error message if it is the first such discrepancy in the record.

Error Messages

Apart from the messages connected with hardware difficulties, which are similar to those for other I/O devices, 2RC gives messages concerning card format. When 2RC is called by CIO, these messages go on the user dayfile in the normal way. But when it is called by 1LJ, the messages have to be saved in some way against the time when the job will come to a control point. This is done by setting up an extra file the first time a format message is needed for a job. In the FNT, this file has the same name as the job file being constructed, and its control point number is 0. But its type is local, not input, and its disposition code is 0. After the job file has been completed and left in the input stack, the extra file will not be noticed by any program (because its disposition code is 0) until 1BJ brings the input file to a control point. Then 1BJ will scan the FNT for a possible extra file with the same name, control point number 0, type local, disposition code 0, and if it finds it, will make its FNT entry into the entry for a file called OUTPUT, with the control point number for the new job, still type local. Thus the format messages that 2RC produced will appear on the first page of the job's ordinary OUTPUT file. The FET for this extra file is put in RA+20B through RA+24B. While being written, the file has name "MESSAGE" and belongs to the "READ" control point. Then 2RC changes the FNT entry so that its name is the same as that of the job file, and its control point number is 0. But there can be no confusion between files, because the job file belongs to the "READ" control point until 1LJ and 2RC are through with it.

The buffer for the extra file is in RA+3100B to RA+3300B, the end of the field length for the READ control point. If the buffer gets full, we do not for simplicity's sake empty it by writing; we merely fail to write out any format error messages thereafter. As the format error messages are only 3 CM words long, this allows for 40 of them, which ought to be enough.

When 2RC finds a format message that calls for an abort (a bad check sum or an unrecognizable binary format when not in 80-column status), then if we are at a job control point, it can be aborted in the normal way. But if 2RC is at the READ control point, it zeros RA+25B, which 1LJ has initially set non-zero, and which is not otherwise used, and 1LJ will pass this pre-abort signal on and put it in the job file FNT entry for 1BJ to see later.

The format error messages are:

```
MODE CHANGE RC.00,CD.0000
SERIAL CHK.  RC.0000,CD.0000 HOLL.CHECK RC.0000,CD.0000
CKSUM ERROR RC.0000,CD.0000 FORMAT ERR. RC.0000,CD.0000
```

The two numbers are the record number, in decimal, and the card number, in decimal.

MODE CHANGE means that the record type has changed within a record. This message is given only once per record. The record type is set to 0 if the first card of a record is binary, or to 1 if it is Hollerith. If a card of the other type occurs, the record type is changed. When 80-column status begins, the card that begins it has 7777B in exactly one of columns 2 through 80, and the number of that column becomes the record type. If this happens at the beginning of a record, no mode change is considered to occur. When a copy of the card that began 80-column status is read, the record type is set to 177B, and this is not considered a mode change. But if the next card is not an end of file or end of record, it will change the record type to something between 0 and 80, and this will be a mode change.

SERIAL CHK. means that the nth card of a record is a normal binary card, but its serial number in column 79-80 is not n. This is given only once per record.

HOLL.CHECK means that a Hollerith card has been read on which at least one column contained an invalid combination of punches, and was read as blank. This is given for every such card.

CKSUM ERROR means that a normal binary card with a bad checksum in column 2, and no 4-punch in column 1, has been read. This will cause an abort, but the rest of the card file will be read, and format messages will still be given for the rest of the file as necessary. CKSUM ERROR itself will be given for every such card in the file.

FORMAT ERR. means that a binary card with no recognizable format has been read while not in 80-column status. This will cause an abort, but the rest of the card file will be read, and format messages will still be given for the rest of the file as necessary. FORMAT ERR. itself will be given for every such card in the file.

If either of the last two messages is given, causing an abort, the last format message will be "JOB PRE-ABORTED". Unless 2RC is at a job control point, in which case the system will give an ordinary abort message.

Narrative

As the 2RC assembly listing is copiously commented, we shall try to keep this short.

2RC begins by testing the end-of-job flag in the FNT. If this is 1, the last call to 2RC caused us to read an eof card when the status was not already eor. We then set the flag and proceeded as if for eor. Now we must proceed as for eof without reading another card, so we set the status to eof, zero the record count and record type fields in the FNT, call subroutine DUMP to write out the accumulated format error messages, if any, and set the pre-aborted flag in RA+25B if necessary. But if 2RC is at a job control point, DUMP does nothing. Then exit from 2RC back to 1LJ or CIO.

If the end-of-job flag is not set, we come to CRD1 and call the initializing subroutine PRS. This is the only place where it is called. PRS formats the 6681 select code and card connect code, using the EST entry in D.EST through D.EST+4. Then it formats the card reader EST ordinal number, which is found in bits 24-35 of the input register, and inserts it in the message at MSGA.

Now we check the buffer pointers, and if there is not room to add 16 words to the buffer (the most that could come from one card), we exit back to 1LJ or CIO. Otherwise, call subroutine RES to reserve the channel, select the 6681, and connect the card reader and instruct it to read the next card in binary regardless of its format. Then zero the word at control point +35B, to clear from the B-display any previous hardware trouble messages.

Now we are at STS1. We read the card reader status, and store it at STAT. If bit 1 is 0, go ahead to CRD6, otherwise go to BSY. At BSY we send a 0 function to clear the card reader, then call subroutine REL to deselect the 6681, drop the channel, and pause for storage relocation. Then come back to CRD5 to try the card reader again. At CRD6 we pick up the card reader status again. If bit 0 is 1, we store 1 at LSTAT, and go ahead to ROC1. Otherwise, zero LSTAT and go to NRD. At NRD we check bit 10 of the status word. If this is 0, the card reader is merely not ready, and we call subroutine MSG to put the message "NOT READY" on the B-display, and then go to BSY as above. Otherwise, the situation is more serious. We check bit 0 in LSTAT. If it is 0, we have already given the message and go to CME1. Otherwise we zero LSTAT, and call subroutine MSGD to put the message "COMPARE ERROR" on the B-display and in the system dayfile, but not in the control point dayfile. Then call subroutine REDCNT to reduce by 1 the card count in the FNT, and subroutine BACKUP to return D.IN and D.IN+1, and the IN pointer in the FET, to what they were before the last time a card was stored in the buffer. Remember that we have not yet read anything on this trip through the main cycle; it is the preceding card that is giving a compare error, and we have to un-read that card. Now at CME1 we put the message "RE-READ LAST CARD" in the B-display, and go to BSY as above. The card reader stopped itself because of this compare error, so only one card has to be re-read.

When we get to ROC1, we have obtained a reader ready status, and stored 1 at LSTAT. Now we read a card, in binary, into BUF through BUF+79, one column per cell. Then increase by 1 the card count in the FNT. At ROC5, we read the 6681 status. If bit 2 is 1, we are in trouble again, and go to XPE. Otherwise, call subroutine RELCR to release everything and then check whether the card had 7 and 9 punches in column 1. If so, go to CRD71 to treat it as a binary card; otherwise continue at CRD7, to treat it as a Hollerith card.

At XPE, we call subroutine MSGD to put the message "6681 XMSN PARITY ERROR" on the B-display and in the system dayfile but not the control point dayfile. The card reader has not stopped itself, as the trouble is in the 6681, so we shall have to set a pause bit, and not continue reading till the operator has cleared it. We have to reread 2 cards, of which only the first has been transferred to the PP by the 6681, but badly as it

seems. Now we clear the channel. Next we call subroutine BACKUP to back up the IN pointers to where they stood when the preceding card was about to be moved to the CM buffer. Probably this should not be done. But the call to BACKUP here is not known to have caused trouble so far, and it is not easy to force a 6681 parity error so as to settle the question by trying it out. Now we call subroutine REL to deselect the 6681 and drop the channel, and branch to ERR. This is the only way to get to ERR. At ERR, we call subroutine MSG to put message "RE-READ LAST 2 CARDS, TYPE GO" on the B-display, and then set bit 12 of the word at RA+0 to 1, as a pause flag. Now we call subroutine REDCNT twice, to reduce the card count in the FNT by 2. Probably REDCNT should only be called once, as the two cards we are about to re-read include the one the PP has just received, and the next one, which the PP has never actually read. To settle the question is not easy, as noted above. Then we reserve and release the card reader repeatedly until bit 12 of RA+0 has been zeroed, by the operator typing "n.GO" at the console after backing up the card input by two cards. Then go back to CRD5 to start over.

If the 6681 parity error bit was not found, we go to CRD7 or CRD71, as explained above. From here on, the assembly listing is heavily commented. If we have just read an end of file card, we go to EOF. Now just before 2RC was called, 2BP was called, and it moved the last status of the FET to BS. If this does not show end-of-record status, or if the buffer is not now empty, we go to BIN1 to set the end-of-job flag in the FNT. Then we go to EOR to pretend we have an end-of-record card, first zeroing BUF+1 and BUF+2 to produce an eor level number of 0. But if the last status was eor, and the buffer is empty, we now set the status to end of file, call subroutine DUMP to write the accumulated format messages and set the pre-abort flag if necessary; then zero the record count and go to BIN3.

When we read an end-of-record card, or want to simulate it as described in the preceding paragraph, we come to EOR. We increase the record number in the FNT by 1, and zero the record type in the FNT. Then reduce the level number from columns 2 and 3 of the card, in BUF+1 and BUF+2, and insert this in the FNT. Then go to BIN3.

We come to BIN3 after setting the FNT properly on reading an eof or eor card. We clear the card count in the FNT, and zero MODCHY and NUMCHY, so that the first mode change or serial number check in the next record will produce a format message. Then go to CRD to exit from 2RC back to 1LJ or CIO.

When we read a special card that begins 80-column status, we arrive at CRD75A+2, where we set the record type to something between 2 and 80, and go back to CRD5 to read the next card.

When we read a special card that ends 80-column status, we arrive at SPMODE+7, where we merely set the record type to 177B and go back to CRD5 to read the next card.

When we read any card while we are in 80-column status, we go to CRD73, where we set WC the word count to 16, put BUF the start of information in the A-register, and go to XFR to move data to the CM buffer.

When we read a binary card out of 80-column status, and have found it has normal format with a 4-punch in column 1 or a correct checksum, we arrive at CRD80. We test NUMCHY; if it is not zero there has already been a serial number check for the record and we go on to CRD90. If it is zero we match the serial number in the card with the card count in the FNT. If they do not check, we set NUMCHY non-zero and call subroutine UMES for a format error message. In any case, we have already put the number of words of information, taken from column 1, in WC. Then we put the address BUF+2, where the information begins, in the A-register and go to XFR to move information to the CM buffer.

When we have read a Hollerith card out of 80-column format, and have translated it into display code beginning in BUF, we get to HOLA. The address of the rightmost non-blank byte, or the address BUF if the whole card was blank, is in D.Z1. We divide this number by 5, discard the remainder, and add 1 to get the number of CM words that have to be moved. Then read a word of five zero bytes to follow immediately the byte whose address was in D.Z1. From one to five of these will go into the CM buffer to terminate the card image properly. We have the number of CM words in WC; we put BUF, the address where the translated card image begins, in the A-register, and we go to XFR to move data into the CM buffer.

At XFR we store the address where the information from the card begins at XFR2. Then save the IN pointer in BACKA+1 and BACKB+1, so that if subroutine BACKUP is called, the IN pointer can be backed up one card. We know there are at least 17 empty words in the buffer, as we checked this before reading the card, so there is no worry about overflowing it. Find out how many words are free between IN and LIMIT, and put the smaller of this and the word count from WC into D.Z1. Now if WC contains zero (this should never happen) go straight to XFR4 without moving anything. Now put the difference between D.Z1 and WC into D.Z2; this is the number of words that will have to be put at the beginning of the buffer if the card information cannot fit between IN and LIMIT. Now move the number of words given in D.Z1 from the beginning address of the card image information to the buffer address D.IN and D.IN+1 point to. Now if D.Z2 contains zero, everything has been moved, and we go to XFR4. Otherwise move the number of words given in D.Z2 from the beginning of the card image plus five times the number in D.Z2 (number of bytes already moved) into the buffer beginning where D.FIRST and D.FIRST+1 point.

At XFR4, we have moved the card image into the CM buffer, but have not advanced the IN pointer either in PP or in the FET. We check the status in the FNT. If this is eor or eof, exit from 2RC back to 1LJ or CIO without advancing the pointer (it does not appear that this can happen. The code must be left over from a version in which we branched to XFR with 0 in WC for eor or eof, after setting the status in the FNT. Then we would come to XFR4 without moving anything which scarcely seems possible now, and exits from 2RC at this point). Now advance D.IN, D.IN+1, and the FET IN pointer according to the number of words in WC. Then we reset D.OUT and D.OUT+1 according to the FET OUT pointer, in case somebody else emptied some of the buffer while we were working. Then return to CRD2 for the next card. Note that we returned to CRD5 for the next card when we have not actually read any information, but here we have to go back to CRD2 to begin by checking that there are at least 17 empty words in the CM buffer.

Subroutines

BACKUP

This restores D.IN and D.IN, and the FET IN pointer, to where they were just before the last time a card image was moved into the CM buffer. The old values were saved in BACKA+1 and BACKB+1 just before moving data in the section of the main cycle beginning at XFR. There may be a possibility of trouble here, if someone has been taking data from the buffer and moving the OUT pointer while 2RC was at word. This will not happen at the READ control point, as 1LJ waits for 2RC to finish before it does anything with the information. But at a job control point, a user might call CIO to read a local file assigned to a card reader, without requesting recall, and then use the data in the buffer as fast as it arrived.

MSGD

This is called, with the starting address of a message in the A-register, to move the message to the control point for the B-display, and then call subroutine R.DFM to write it on the system dayfile but not the control point dayfile.

MSG

This is called, with the starting address of a message in the A-register, to move the message to the control point for the B-display. The message is copied into MSGA+3 ff., so that it will be prefixed by "CRnn". Copying stops on a zero byte. Then MSGA through MSGA+14 are copied to control point +35B through 37B.

RES

This is called to reserve the channel for the card reader; to fill the channel number into the instructions listed at RESA, if they do not contain it already; to select the 6681, to send function code 1 to the card reader, so that it will read in binary regardless of the actual format of the next card, and to go into subroutine CFR to check acceptance of this function. If the response from CFR is non-zero, subroutine REL has already been called to release everyting, and we start over in subroutine RES. If the response from CFR is 0, we exit from RES.

REL

This is called to deselect the 6681, drop the channel whose number was stored by subroutine RES at CHAN, and pause for storage relocation. We drop the PP if the error flag is non- zero.

RELCR

This is called to clear the card reader. We send function code 0 through subroutine CFR, which passes it to the card reader. If the response from CFR is non-zero, we exit immediately from RELCR, as subroutine CFR has already called subroutine REL. Otherwise, we call subroutine REL before exiting from RELCR.

CFR

This is called to pass a function code, which is in the A-register on entry, to the card reader. Then we read the 6681 status. If bits 0, 1, and 2 are zero, we exit from CFR with 0 in the A-register. If bit 1 or 2 is 1, but bit 0 is 0, we send the message "CR nn REJECT" to the B-display and the system dayfile; then clear the channel, release the channel and the 6681, and exit with 1 in the A-register. If bit 0 is 1, we send the message "CR nn FUNC XMSN PARITY ERROR" to the B-display and the system dayfile; then wait 1 second, clear the channel, release the channel and the 6681, and exit with 1 in the A-register.

UMES

This is called with the starting address of a message about a card format error in the A-register. The record and card numbers are taken from the FNT, translated to decimal, and inserted after COUNTS. Then the message is copied to the cells before COUNTS. Then if 2RC was called by 1LJ, we put the starting address of the combined message in the A-register, and call subroutine WUMES to put it in the special buffer for the future output file. Otherwise we call subroutines MSG and R.DFM to put the combined message in the B-display, the system dayfile, and the control point dayfile.

WUMES

This is called with the starting address in the A-register of a message that is to be put in the special buffer, for the future output file. We get the pointers from RA+20B through 24B. If the buffer had not at least 4 empty words, do nothing. Otherwise, advance the IN pointer by 3, and copy the 3 CM words of the message into the buffer.

CONVERT

This is called to convert a binary number, in the A-register on entry, to a 4-digit decimal number, and add it to the display code "0000", which is already in two bytes, of which the address of the first is in D.Z1.

ABORT

We come to abort, with the address of the appropriate message in the A-register, when we have found a binary card with an unrecognizable format or a bad checksum. We immediately call subroutine UMES to handle the message. Then see whether 1LJ or CIO called us. If 1LJ, store zero at RA+25B as a pre-abort flag for this job file, which 1LJ will pass to 1BJ in its FNT entry. Then go back to CRD2 to continue reading the file for the sake of further error messages. If called by CIO, go to ABORTA, set the status of the FNT to complete, and abort the job.

DUMP

This is called when 2RC is about to give control back to 1LJ or CIO, after an end of file. If CIO, DUMP does nothing, but if 1LJ it first checks the pre-abort flag in RA+25B. If this is set, i.e. zero, subroutine WUMES is called to add the pre-abort message to the file error messages. Then the routine is exited if the IN pointer of the FET at RA+20B is equal to the field length-200B, i.e. to the FIRST pointer of the special buffer. Otherwise, we write the name "MESSAGE" and the function code for write-end-of-record into the first word of the FET; then add 1 to the "L" field of the FIRST pointer, indicating a non-minimal FET so that when an FNT entry is created for the message file, its address will automatically by put into the LIMIT pointer word. Then call CIO to another PP to process this FET, and wait till the FET shows completion. Now we find the FNT entry, according to the pointer in the FET LIMIT word, and see that the FET pointer in the FNT points, in turn, back to RA+20B. Then it alters the control point number in the FNT entry to 0, and the name of the file to the name of the job file, which (and not "READ") is found at control point+21B. As we have both created, and released to control point 0, this FNT entry while 1LJ was waiting for us, 1LJ is not even aware of this future output file even though it has the same name as the job file whose termination we are about to signal. Then exit from DUMP.

2TJ - TRANSLATE JOB CARD

2TJ is a relocatable overlay which is called to scan and interpret a job card whose image is contained in the callers field length. 2TJ checks the job card for validity, obtains a sequence number for the job and returns to the calling program the parameter values specified on the card. It must be called by any routine which makes entries into the input queue for a new job. This would include JANUS, RESPOND and EXPORT/IMPORT.

Entry Conditions

At entry the job card image must be stored in central memory within the field length for the control point. The following direct cells must also be set up:

D.RA	Reference address of the control point
D.FIRST, D.FIRST+1	The relative address of the job card image (17 bit address, right-justified).

Programs calling 2TJ must provide the first word address of a parameter table (2TJ passes job card parameters back to its callings routines in a 15 byte table) in direct cell D.JPAR:

D.JPAR	FWA of parameter table.

Normal Exit

If the job card is valid, 2TJ will return all job card parameters in a 15 byte parameter table with first word address stored in D.JPAR. The parameter table will be in the same format as an input FNT entry.

Error Exit

If a job card error is detected, 2TJ will set byte C.JABT (bits 3-5) to a value of 6B. Job field lengths, dependency count and dependency variable, tape unit requirement count will be zeroed. The priority will be set to the highest non-fixed value. If the name on the job card was valid D.JNM, D.JNM+4 will contain the job name as in a normal exit, otherwise D.JNM, D.JNM+4 will contain the name ERROR followed by 2 sequence characters.

General Description of the Code

The function performed by 2TJ is essentially quite a simple and straight - forward one. The code, however, is quite complex because an attempt was made to be very general. 2TJ can handle job cards in either of two formats (SCOPE 2 or SCOPE 3); parameters on the job card can be treated as octal or decimal quantities; and 2TJ can be made to compute priority based on the other job card parameters. Each installation can control the above parameters by inserting macro calls into the normally empty SCPTEXT macro named JOBCARD.

JOBCARD Macro

Normally octal parameters and a SCOPE 3 format (mnemonics followed by parameter values) are assumed. Defining the symbol SCOPE 2.0 in JOBCARD will cause 2TJ to interpret job cards as SCOPE 2 format (parameter values separated by commas). SCOPE 2.0 job cards will not be able to use job dependency and scheduling dependent on tape unit availability. Calling the macro DECIMAL

FIELD (where FIELD is one of the following mnemonics: CM, EC, T, P) will cause the parameter indicated to be treated as a decimal quantity. Calling the macro

```
        WEIGHT   FIELD,RELATION,VALUE,ADDITIVE
```

in JOBCARD will cause addition code to be generated in 2TJ. This code checks to see if the specified RELATION (GE or LE) exists between the specified field (T, CM or EC) and the VALUE. If it does the ADDITIVE is added to the job priority.

3RP--Close Reel Processing

General Description

The purpose of 3RP is to perform the CLOSE REEL function for magnetic tapes. The routine is called whenever a tape driver encounters end-of-reel (EOR) or end-of-information (EOI). End-of-reel is the reflective spot; end-of-information depends on tape type and label type.

A. SCOPE Standard Tapes

Since the WRITEF request results in a logical end-of-file mark, a physical end-of-file mark can only occur as part of an EOF label, indicating end-of-information.

B. Unlabeled X, S, and L tapes

The WRITEF request results in a physical end-of-file mark. A backward motion following a write or open write causes four physical end-of-file marks to be written. The system cannot recognize end-of-information in this case. The user must know how many files are on his tape and make an explicit CLOSE REEL request at the appropriate time; for these tapes, the driver calls 3RP only when end-of-reel is encountered.

C. Labeled X, S, L tapes

Since a WRITEF request results in a physical end-of-file mark, a tape read driver cannot determine whether this means simply end-of-file or whether it is part of a trailer label and therefore implies end-of information. 3RP is called to read the next PRU and check for a label.

General Logic Flow

A tape driver calls 3RP when a file mark or end-of-tape is encountered. If a file mark was read, 3RP calls 4LB to read the next record and test for a trailer label. A trailer label is an 80 character record beginning with the display code characters EOF or EOV for a standard labeled tape and EOF or EOT for a Y-labeled tape. If a label is not found, 3RP backspaces, increments the PRU count, clears the EOI bit, and sets end-of-file in the FNT. If EOR is off or the UP bit is on, 3RP exits. Otherwise end of reel processing is performed as follows: the next record is read and checked for a valid trailer label.

A. Label unrecognizable -
A dayfile message is displayed with a pause for the

operator to type GO or RECHECK.

B. EOF label read -
The tape is positioned in front of the file mark preceding the EOF label record, EOI is set in the FNT, and 3RP exits.

C. EOV label read -
The tape is positioned in front of the file mark preceding the EOV label record, EOI is cleared and EOR is set. If the UP bit is on, 3RP exits. If the last operation was write or open write, 3RP writes a file mark, a label record, and two more file marks on a standard tape and four file marks on an X, S, or L tape. The tape is unloaded, the PRU count is set to -3 for a labeled tape, +0 for an X tape, and -0 for an unlabeled non-x-tape. Then 3RP displays the end-of-tape message. If the FNT has 2 EST ordinals, they are swapped; and if there is a MFN FNT entry, its primary and secondary EST ordinals are also swapped. Open is called in another PP and 3RP loops until the EOR bit in the FNT is cleared (by OPE), then exits. The exit from 3RP is either to return to the user program or to reload the driver (see Exit Information).

Environment

A. Other routines used

6WM--to write an error message
4LB--to process standard labels
4LC--to process non-standard labels
1xy--the one level overlay which called 3RP

B. Other routines calling 3RP

1RS--When file mark or end of reel is encountered
1MT--When file mark on end of reel is encountered

C. Usage of the routine

3RP is not reentrant. The driver which called 3RP is reloaded over 3RP or the PP is dropped.

D. Residence

3RP can reside on disk or in central memory

Interface with other Routines

A. Entry Information
D.FNT+9 is used by the tape drivers to indicate

whether a file mark was read {EOI bit set} or end of reel was reached {EOR bit set} or both {EOR and EOI set}. Bits 0-8 of D.FNT+9 and the PRU count in PP low core must be updated so that if 3RP finds no label it can increment this PRU count, complete D.FNT+9, and return to the user. Low core cells must be set as follows:

D.FNT through D.FNT+9=FNT words 2 and 3
D.BA through D.BA+4=FET word 1
D.PPIRB through D.PPIRB+4=Input Register
D.RA = Relative address
D.FL = Field length
D.FA = FNT word 2 address
D.CPAD = Control point address
D.PPIR = Input Register address
D.PPOR = Output Register address
D.PPMESI = Message Buffer address

B. Exit Information

If the driver is loaded after 3RP is finished, the low core cells given as input are unchanged except possibly the code and status in D.FNT+9 and the primary and secondary EST ordinals. The cell used for last buffer status {45} and the buffer pointers in low core are left unchanged, and the EST entry is read from central memory.

If 3RP is called with EOR only {data record on the reflective spot}, after swapping reels the driver is reloaded except for READ and RPHR requests which return to the user.

If 3RP is called with EOI set, the status, file position, and exit are shown in the following tables.

READ, RPHR, READN, READSKP

	UP bit on	UP bit off
File mark = eof	1. Set eof, clear EOI, increment PRU count. 2. Position tape after file mark. 3. Complete request and return to the user	1. Set eof, clear EOI, increment PRU count. 2. If EOR was set, swap reels. Otherwise position after file mark. 3. Complete request and return to user.
File mark is part of EOF label	1. Set EOI 2. Position before file mark. 3. Complete request and exit to user	1. Set EOI 2. Position before file mark 3. Complete request and exit to user
File Mark is part of EOV label	1. Clear EOI, set EOR 2. Position before the file mark 3. Complete request and exit to user	1. Clear EOI, set EOR 2. Swap reels 3. Reload driver

SKIPF

	UP bit on	UP bit off
File mark = eof	1. Set eof, clear EOI and EOR. Increment PRU count. 2. Position tape after the file mark 3. If EOR was set, complete the Request and return to the user. Otherwise decrement the skip count and reload the driver	1. Set eof, clear EOI and EOR. Increment the PRU count. 2. If EOR was set, swap reels. Otherwise position after the file mark. 3. Decrement the skip count and reload the driver.
File mark is part of EOF label	1. Set EOI and clear EOR. 2. Position before file mark. 3. Complete the Request and return to user.	1. Set EOI and clear EOR 2. Position before file mark. 3. Complete the request and return to user
File mark is part of EOV label	1. Clear EOI, and set EOR. 2. Position before file mark. 3. Complete the request and return to the user.	1. Clear EOI and set EOR 2. Swap reels 3. Reload the driver

Interface of 3RP with Label Processing Routines

For a standard labeled tape, 3RP calls 4LB to read or write labels and reposition the tape. For a non-standard tape, 4LC is called. The 4LC submitted for SCOPE 3.1.6 is for 3000 labels, but the intent is that a user can write his own 4LC if he wishes to process another label type. A restriction on the format of labels which can be processed without modification to 3RP is: a trailer label must consist of one physical record separated from the user data by a file mark.

In order to interface with 3RP, 4LC routines should be writtten as follows:

1. 4LC should be reentrant so that 3RP does not need to reload it for each entry.
2. 4LC should use the low core cell 46 for a communication word {CW} with input options of
   - 0 for Write volume trailer
   - 7 for write file mark
   - 10 for read and check contents of label
   - 14 for read and don't check data
   - 31 for unload
   - 32 for backspace

   Exit information which should be returned by 4LC in CW after a read request {10 or 14} is
   - 0 if volume trailer read
   - 1 if file trailer read
   - {CW} 4 if record not recognized or if label information is in error.
3. 4LC should not alter the low core cells 20-36, 40-45 50-67, 74-77

5DA/7DA - PRIVATE DISK PACKS

5DA is a PP overlay, always loaded at 5000B, which does all the reading, writing, and checking of disk pack labels except:

a. what is done during dead starting.

b. the writing of blank labels on unloaded packs in response to BLANK type-ins. 5DA checks the existing pack labels, but 1BT writes the new ones.

c. the writing of labels on private packs before they are unloaded at job termination. This is done by 1EJ.

7DA is another PP overlay containing nothing but error messages for 5DA to issue. 5DA loads it into the space that is otherwise used for reading or writing a PRU to or from a pack label.

Entry Information

When 5DA is called as a subroutine, bits 0-11 of the A register contain 1, 2, 3, 4, or 5:

1. 5DA has been called by 1MH, called in turn by DSD, responding to an UNLOAD type-in. Bits 12-17 of the A register contain the EST ordinal of the disk pack to be unloaded.

2. Same as 1., except that the type-in was DEVADD, and the EST ordinal is of the disk pack to be made public.

3. 5DA has been called by 1AJ in response to an RPACK control card, or by REQ in response to a REQUEST control card in which the second parameter is "PK". The first parameter from the card is left justified with zero fill in D.BA through D.BA+4; the second parameter is in D.FIRST through D.FIRST+4, and the third is in D.FNT through D.FNT+4. However, when the second parameter of a REQUEST card is PK, 0001B is found by 5DA in D.FIRST. If a parameter is not present on a card, the corresponding cell, D.BA, D.FIRST, or D.FNT contains 0 when 5DA is called.

   The first parameter on an RPACK card must be the wanted pack name. Either the second or the third must be "E" or "N". If "N", no other information is taken from the control card, and 5DA is to start off an empty private pack with that name. If "E", the remaining parameter on the card, if any, is the visual pack number. 5DA is to bring an existing private pack of that name to the control point and check the visual pack number if specified on the card.

   If the second parameter on a REQUEST card is "PK", the first parameter must be a file name not already in use for this job, and the third parameter must be the name of a private pack already assigned to the control point.

4. 5DA has been called by 1AJ in response to a REMOVE control card. The first parameter from the card is in D.BA through D.BA+4; this must be the name of a file at the control point, assigned to some private pack, which is to be evicted, removed from the job, and removed from the list of files on the pack. If D.EST does not contain zero, there is a second parameter on the card, found in D.EST through D.EST+4. This must be the name of the private pack to which the file is assigned.

5. 5DA has been called by 1BT, which was called by a type-in of "n.BLANK,uu." where uu is the EST ordinal of a disk pack, which is to be blank labeled. Then the EST ordinal is found by 5DA in D.PPIRB+4, bits 0-11 of the input register. 1BT has already got the pack assigned to the control point.

Exit Information

This is classed under 1 to 5, as above, according to the value in bits 0-11 of the A register on entry.

1./2. No exit information, as 5DA drops or aborts the control point.

3./4. If successful, 5DA returns to 1AJ or REQ with 0 in the A register; otherwise, the A register on return contains the address of an error message that 1AJ or REQ must send to the dayfile.

5. If the pack has a blank label or an unrecognizable label, 5DA returns to 1BT with 0 in the A register, indicating that the pack may safely be blank labeled. If the pack has a private label, 5DA has read the first PRU into LABEL through LABEL+477B, and returns to 1BT with the address LABEL in the A register, so that 1BT can format a message to the operator including information from the label. In either case, the pack is still assigned to the control point.

Other Programs Called

5DA calls overlay 7DA whenever is has to issue an error message. It also calls 1SX to another PP when it is loading an existing private pack and needs more RBT space. 5DA recalls 1AJ with 5 seconds delay, after setting the flag telling it to process the same control card again, when it is waiting for an ASSIGN and/or VRN type-in from the operator during the processing of an RPACK control card.

Messages

TYPE IN VISUAL PACK NUMBER.

This message, at location VREEL, is put on the B display after the operator has assigned a private pack in response to an RPACK (pname,N) control card, but before he has given a VRN type-in with the visual identification number.

(uu ASSD.)

This message, at location GASMSG, is put on the dayfile as soon as an operator assignment, in response to an RPACK control card, has been picked up.

(VRN 0000xxxxxx)

This message, at location VRMSG, is put on the dayfile as soon as the operator VRN type-in has been picked up.

These two messages are issued before 5DA has actually looked at the assigned disk pack: so they may be superseded by an error message in the dayfile, after which either the job will be dropped or a new assignment and VRN message will appear in the dayfile.

The remaining messages are all in overlay 7DA.

BAD LABEL.

This message is put on the B display and the dayfile when an unrecognizable label is read on a disk pack, in response to an RPACK control card or a DEVADD type-in, or when a recognizable label with wrong name is read in response to an RPACK (pname,E) control card. For DEVADD, the control point will be aborted. For RPACK, the system will wait for the operator either to drop the control point or to assign another disk pack drive, or the same drive with a different pack on it.

DUPLICATE PACK NAME

This message is sent to the dayfile by 1AJ when the pack name on an RPACK control card is the same as the name of a pack already at the control point. The job is aborted by 1AJ.

DUPLICATE FILE NAME

This message is sent to the dayfile by 1AJ when a.) an existing private pack is being attached to the control point in response to an RPACK control card, and its label names a file with the same name as a file already at the control point. b.) a REQUEST(fname,PK,pname) control card contains a file name the same as the name of a file already at the control point. In either case, the job is aborted by 1AJ.

NO SUCH PRIVATE PACK

This message is sent to the dayfile by 1AJ when a REQUEST or REMOVE card names a private pack that is not at the control point. The job is aborted.

63 FILES ALREADY ON PACK

This message is sent to the dayfile by 1AJ when a REQUEST card calls for a new file to be initiated on a private pack, but the pack already has the maximum number of files assigned to it. The job is aborted by 1AJ.

THE FNT IS FULL

This message is sent to the dayfile by 1AJ whenever 5DA is unable to put a new private pack or private pack file into the FNT because it is already full. The job is aborted.

PACK NOT PUBLIC AND EMPTY

This message is sent to the dayfile by 5DA when an UNLOAD type-in references a disk pack that cannot be unloaded because its EST status is not 4000B. The control point is then aborted.

THIS PACK STATUS NOT UNLOADED

This message is sent to the dayfile by 5DA when a DEVADD type-in references a disk pack that cannot be made public because its EST status is not 4040B. The control point is then aborted. The same message is put on the B-display and the dayfile by 5DA when an ASSIGN type-in, following an RPACK control card, references a disk pack whose EST status is not 4040B. In this case, the system waits for the operator to drop the job or try another assignment.

CANT UNLOAD PACK

This message is sent to the dayfile by 5DA when monitor refuses a request to set the EST status of a disk pack to 4040B. Something is wrong with the system. The control point is aborted.

CANT GET PACK ASSIGNED TO C.P.

This message is sent to the dayfile and the B-display by 5DA when monitor refuses a request to set the EST status of a disk pack to 402pB, p being the control point number. If there is nothing wrong with the system, the explanation must be that the pack has been taken from under 5DA's nose for some other public or private use. If this is in response to an RPACK control card, the system waits for the operator to drop the job or try another assignment. Otherwise, the control point is aborted.

CANT SET PACK AVAILABLE

This message is sent to the dayfile by 5DA when monitor refuses a request to set the EST status of a disk pack to 4000B. Either something is wrong with the system, or some other PP has taken the pack from under 5DA's nose. 5DA must have been called for a DEVADD type-in, and it now aborts the control point.

PACK LABEL NOT BLANK

This message is sent to the dayfile and the B-display by 5DA when it reads a private pack label from a disk pack that ought to have a blank label. If this is in response to a DEVADD type-in, the control point is then aborted. If to an RPACK(pname,N) control card, the system waits for the operator to drop the job or try another assignment.

BAD PACK NAME IN REQUEST

This message is sent to the dayfile by 1AJ when an RPACK or REQUEST control card contains a pack name with a bad format, or a REMOVE card names a file that resides on a private pack whose name is different from the pack name given on the card. The job is aborted by 1AJ.

NOT DISK PACK

This message is sent to the dayfile and the B-display by 5DA when a DEVADD, UNLOAD, or BLANK type-in, or an ASSIGN type-in responding to an RPACK control card, gives an EST ordinal that does not correspond to a disk pack. For ASSIGN, the system then waits for the operator to drop the job or try another assignment. In the other cases, the control point is dropped.

EST ORDINAL TOO HIGH

This message is sent to the dayfile and the B-display under the same conditions as the preceding one, when the type-in gives an EST ordinal larger than the length of the EST, as specified in P.EST.

NO RBR FOR THIS PACK
This message is sent to the dayfile and B-display under the same conditions as the preceding two, when the type-in gives an EST ordinal that specifies a disk pack, but no RBR referring to that pack by its EST ordinal can be found. Something is wrong with the system, or at least there is an omission in the RBR tables in CMR.

WRONG VRNO
This message is sent to the dayfile and B-display by 5DA when the operator type-in in response to an RPACK(pname,E,vrno) control card names a pack whose label contains the right pack name but the wrong visual identification number. The system waits for the operator to drop the job or try another assignment.

NO SUCH FILE
This message is sent to the dayfile by 1AJ when a REMOVE control card names a file that is not at the control point. The job is aborted.

FILE NOT ON PRIVATE PACK
This message is sent to the dayfile by 1AJ when a REMOVE control card names a file that is at the control point, but is not assigned to any private pack. The job is aborted.

PACK FILE COUNT ALREADY 0
This message is sent to the dayfile by 1AJ when a REMOVE card names a file that is on a private pack, and the pack has the name specified on the card, if any; the file has been destroyed, but on attempting to reduce by 1 the file count in the FNT entry for the pack, 5DA found the count already zero and left it unchanged. Something is wrong with the system. The job is aborted.

SUCH NAMES FORBIDDEN TO PR. PACK FILES
This message is sent to the dayfile by 1AJ whenever a REQUEST(fname,PK,pname) control card contains a file name which is one of those in the list at NAMTAB in 5DA. The job is then aborted by 1AJ. These are the names that automatically bring disposition codes to their files at job termination; but no private pack file is allowed to have a special disposition code, because the file will not be available to the system after job termination.

NO E OR N ON CTRL CARD
This message is sent to the dayfile by 1AJ whenever an RPACK control card does not have either "N" or "E" as its second or third parameter. The card is not analyzed further, and the job is aborted.

BAD FILE NAME ON REQUEST
This message is sent to the dayfile by 1AJ when a REQUEST(fname,PK,pname) or a REMOVE control card gives a file name with a bad format. The job is aborted.

Narrative

5DA is fairly tight for space, as 7200B-7700B must be available for reading and writing label PRU's. So immediately on being entered, it branches to PRESET, which is within the area that can be overlaid by a PRU if necessary. The A register is immediately stored in D.TR; its value is 1 for UNLOAD, 2 for DEVADD, 3 for RPACK or a REQUEST that initiates a new file on the pack, 4 for REMOVE, or 5 for BLANK. This value in D.TR remains unchanged, and can be used to determine what sort of function is being done, or as a constant =3 for RPACK and REQUEST. For UNLOAD and DEVADD, the EST ordinal that was part of the type-in will be in bits 12-17 of the A register, and this is stored at ESTO. Then 38, the number of CM words in an RBR table, is stored for convenience in D.SV2; and the control point number is inserted into the call to 1SX at R1SX. Then we branch to the appropriate routine, whose starting address is found in table FIVEDB.

At DEVADD and UNLOAD, we set the job name in the control point to DEVADD or UNLOAD, and then branch to UNDEVE, which has to be in the non-overlayable part of the program, as CKEST is called to verify the EST ordinal; if it finds an error it will call overlay 7DA to provide an error message, and we want to be able to exit from CKEST to a non-overlaid part of 5DA.

CKEST scans the RBR tables for one with the appropriate EST ordinal, and copies the DST ordinal from that table to DSTO. The exit is with 0 in the A register if this is successful. If not, an error message has already been given, and we branch to ABORT to abort the job. If successful, we branch to UNDEVF, in the overlayable part of the code. The EST entry has been read into D.EST ff.; we now check the EST status, which must be 4040B for DEVADD or 4000B for UNLOAD (table UNDEVA.) If the status is wrong, go to UNDEVC, pick up the starting address of the appropriate message from table UNDEVD, and go to ABORTB, where we issue the dayfile message before aborting.

If the status is correct, we pick up the job name from the control point, and add to it the formatted EST ordinal, forming a dayfile message for completion at UDMES. Then branch to UNLA for UNLOAD, or DEVADA for DEVADD; these must be in the non-overlayable part of the program as they involve reading and writing the disk pack label.

At UNLA, call subroutine GET to bring the disk pack to this control point. There is a possibility that the system will, since the EST status of the pack was found to be 4000B, have taken some space on the pack and made its status 4010B. In that case, subroutine GET will have failed to bring the pack to our control point, issued a dayfile message, and aborted. (When GET is called for RPACK or REQUEST it does not abort but exits with non-zero in the A register; the distinction is made by seeing whether D.TR contains 3.) On return from GET, we set up a model of the first PRU of the disk pack containing initial "DEV1", the Julian date from CM resident, and the current RBR table from CM resident. (The address of the RBR table is found by subroutine GRBRAD using the RBR ordinal, which was set by subroutine CKEST. This table will show an empty RBR, except for the first record block, reserved for labelling, and for any "flaw bits" that may have been set.) The checksum is formed by subroutine

CKSUM and then stored in the 40th byte of the label PRU. Then subroutines WRITEP and MK40 are called to write the first PRU on the pack and return it to unloaded status (4040B). WRITEP uses the stack request at ZERQ, which is already complete in 5DA except for the EST and DST ordinals; ESTO and DSTO, already set as explained above, are the appropriate bytes of this request. Finally, at FIVEDC, we issue the dayfile message "UNLOAD uu." and exit from 5DA.

At DEVADA, we call subroutine CZLAB to read the first PRU of the pack and check that it is a zero label, so that the pack can be used by the system without fear of destroying anything. If so, the exit from CZLAB is with 0 in the A register; otherwise, an error message has already been given by CZLAB and we go to ABORT. Next we call subroutine MEST twice, to alter the EST status of the pack from 4040B to 4020B and then to 4000B, which makes it available to the system as a public pack. Probably the intermediate change to 4020B is unnecessary. If either of the calls to MEST is unsuccessful, the exit is with non-zero in the A register, and we pick up the pointer to the message at CANTADD and go to ABORTB to issue it and abort. Otherwise, call subroutine STORBR to write into CM the RBR table from the pack label, and then go to FIVEDC to issue the dayfile message "DEVADD uu." and exit from 5DA.

If the A register contains 5 on entry to 5DA, we branch to BLANK. In this case, 5DA has been called by 1BT, and the EST ordinal is in bits 0-11 of the input register, not in bits 12-17 of the A register. So we copy it into ESTO and then branch to BLANKB, which has to be in the non-overlayable part of the program. At BLANKB, subroutine CKEST is called to ascertain that the EST ordinal really refers to a disk pack, and to find its RBR ordinal and DST ordinal. If CKEST finds an error, it issues an error message, so all calls to CKEST have to be from non-overlayable locations to avoid having the calling part of the program overlaid by 7DA. If this has happened, the exit from CKEST is with non-zero in the A register, and we branch to ABORT.

Otherwise, we call subroutine CZLAB to read the label and check that it is either a blank label or not a recognizable label at all. Note that the case of unrecognizable label is found by subroutine CKLAB, which is called by CZLAB, and that CKLAB will then check D.TR to see if this is a BLANK call to 5DA; if not, an unrecognizable label is bad, but if so it is perfectly good.

If, just below BLANKB, the return from CZLAB is with zero in the A register, the pack had either a blank label or no recognizable label, and we exit from 5DA back to 1BT with zero in the A register; 1BT will go ahead and write a blank label and then return the pack to unloaded status. Otherwise, the pack has a recognizable non-blank label; i.e. it must be labelled as a private pack. 5DA exits back to 1BT with the address at which the copy of the label begins in the A register, so that 1BT can format the information of the label into an operator message and wait for the operator to type GO or DROP before blank-labelling the pack.

If 5DA is entered with 3 in the A register, we branch to PRIV, to do something about a private pack. 5DA has been called by 1AJ in response to a control card of one of three types:

```
RPACK(pname,N)
RPACK(pname,E) or RPACK(pname,E,vrno) or RPACK(pname,vrno,E)
REQUEST(fname,PK,pname)
```

The three parameters are in D.BA, D.FIRST, and D.FNT, except that for the third type of card, the "PK" will be represented by 0001B in D.FIRST. So at PRIV, we check for this case first and branch to PFILE if so. Otherwise, if the second parameter is N or E, we set PRIVF+1 to zero or non-zero respectively, then copy the third parameter (if any) from D.FNT to D.FIRST, and go to PRIVE. If not, but the third parameter is N or E, we set PRIVF+1 zero or non-zero respectively, leave the second parameter in D.FIRST, and go to PRIVE. If neither parameter is N or E, we go to PFILA to exit from 5DA with a pointer to a bad-card message in the A register; 1AJ will issue this message.

At PRIVE, we call subroutine VFN to check the first parameter in D.BA for validity (not more than seven letters and/or digits; first character a letter; no embedded blanks or 00B characters). If the name is bad, VFN will exit from 5DA with a pointer to a bad-name message in the A register, and 1AJ will issue the message. On return from VFN, we must see if the first parameter, a pack name, is the same as a pack name already in the FNT for this control point.

Note that whenever a private pack is assigned to a control point, there will be an entry ("pseudo-entry") in the FNT with this form:

```
VFD     42/400000000000000B+pname,3/3,3/c,12/0
VFD     12/0700B,24/0,12/esto,12/n
VFD     60/1
```

where c is the control point number, pname is the logical name of the pack, esto is the EST ordinal, and n is the number of files currently residing on the pack. As the logical name of the pack, as of any file, must begin with a letter, the leftmost bit of the first word would naturally be 0, but is set to 1 instead to distinguish this type of FNT entry from a real file entry.

So now we set the first bit of the name in D.BA to 1, so that it will coincide with the name in an existing FNT pack entry, and call SKFNT with a pointer to D.BA in the A register. SKFNT, if entered with a pointer rather than 0 in the A register, scans the FNT for an entry at the same control point with the name pointed to, and exits with 0 in the A register if none is found, or the address of the FNT entry if one is found. If none is found, we go ahead to PRIVET, but if one is found, we exit from 5DA with a pointer to a duplicate-name message in the A register; 1AJ will issue this dayfile message.

At PRIVET, we see if there is a device assignment waiting in the control point, and if so go to GETASA. If not, at DELAY, we set the bit in the control point that tells 1AJ to reprocess the same control card, and issue a call for 1AJ with a 5-second delay, and go to END to drop the PP. At GETASA, we store the assigned equipment ordinal in ESTO, which is actually part of a model stack request. Now if PRIVF+1 contains zero, a new private pack ("N") is called for, and we shall need the visual identification to be typed in. Otherwise, an existing pack ("E") and we can skip to GETASB. If the visual identification is needed, we see if it is already waiting in the control point. If so, go to GETASC with the visual identification left-justified in D.FNT ff. If not, copy the message TYPE IN VISUAL PACK NUMBER into the control point B display, and go to DELAY to have 1AJ recalled to process the same card in five seconds.

When we arrive at GETASC, we clear the visual number space in the control point, and then pass to GETASB. At GETASB, clear the equipment assignment byte in the control point area. Then form the message (uu ASSD.) where uu is the EST ordinal, and send it to the dayfile. Then check PRIVF+1 again; if it is nonzero, skip to GETASD, but if zero we have a new visual number, and call subroutine PADVRN to format it; then send the message (VRN 0000nnnnnn) to the dayfile, where nnnnnn is the visual identification number, right justified with display code zero fill.

At GETASD, we call subroutine CKEST to see that the EST ordinal in ESTO refers to a disk pack, and to find its RBR ordinal and DST ordinal. If CKEST finds the ordinal wrong, the exit is with non-zero in the A register, and we return to DELAY to ask, in effect, for a new assignment and visual number if necessary. If the return from CKEST is good, the EST entry has been read into D.EST, and we check that the status is unloaded (4040B). If yes, skip to PRIVFA; if not, call subroutine ERMES to issue the not-unloaded dayfile message and return to DELAY to try for a new assignment.

At PRIVFA, we set a bit in the control point that will tell 1EJ that a least one private pack has been assigned to the job, so that private pack termination procedures will have to be carried out at the end of the job. Then, at PRIVF, branch to PRIVEX for an existing pack("E" on the control card). For "N" on the control card, representing a new pack, we continue downward and call subroutine CZLAB to see that the assigned pack has a blank label. If not, CZLAB will issue the appropriate error message and return with non-zero in the A register, so that we return to DELAY to try for a new assignment. If the return is good, we set up at LABEL ff. the model of a blank label PRU. Characters 1-4 are "DEV1"; characters 5-10 are a copy of the right six characters of the Julian date word in CMR; characters 11-20 are the visual identification number, right justified with display code zero fill; characters 31-38 are the pack name, left justified with zero fill, and with the leftmost bit restored to zero; characters 39-40 are the binary count of files on the pack, which is initially zero. Characters 81-460 already contain the empty RBR table read from the pack by subroutine CZLAB (LABEL+40 through LABEL+229). As the pack had a blank label with an RBR table appropriate to use as a public pack, we now alter the 11th byte of the table from 4000B to 7000B, indicating that not the first one but the first three record blocks have to be reserved for the label, and correspondingly reduce the ninth byte (count of available record blocks) by two. (Possibly in future a flaw table will be effectively included in the RBR of a pack label; if so, the 11th byte might contain flaw bits, and it could not be simply set to 7000B, but a more careful procedure would have to be followed.) The rest of this PRU model can be left unchanged as it came from the blank label model, except that a new checksum must be formed by subroutine CKSUM and stored in the 40th byte. Then we call subroutine WRITEP to write this PRU on the pack. Note that WRITEP uses the stack request at ZERQ after setting the order code to O.WRP+4000B; the EST and DST ordinals have already been filled in (at ESTO and DSTO) and the rest of the request is preset to read or write PRU number zero in record block zero, which is all that 5DA is concerned with except when attaching an existing private pack, for which the code begins at PRIVEX.

We also call subroutine PAKFNT, with 0 in the A register as the number of files on the pack, to set up an FNT entry for the pack with that number in bits 0-11 of its second word. PAKFNT copies into the proper place in CMR the RBR table we have just set up in LABEL+40 through LABEL+229. Then it calls subroutine SKFNT to find an empty slot in the FNT, and if one is found, puts in the proper

entry for the pack, with the format previously described. If SKFNT finds the FNT already full, it exits from 5DA with a pointer to a full-FNT message in the A register; 1AJ will issue this dayfile message. SKFNT also calls subroutine AVOID in case of this failure, but AVOID does nothing unless we aretrying to attach an existing pack; in that case it drops the pack immediately to avoid having its label re-written when the job terminates, as this would effectively destroy the good files on the pack.

Finally, we exit from 5DA with 0 in the A register, indicating success.

At PRIVEX we begin the special processing for an existing pack. It is necessary to get all the existing files on the pack into the FNT, so as to make them available for the job. First we call subroutine CKLAB to see that the pack has a recognizable label. If not, the exit is with non-zero in the A register, and we return to DELAY to try for another assignment and start over. Otherwise, compare the pack name on the pack, now in LABEL+15 through LABEL+18, with the name from the control card, which is in D.BA ff. with its leftmost bit altered to 1. If they do not match, we go to PRIVXB, call subroutine ERMES to issue a bad-label dayfile message, then subroutine MK40 to return the pack to unloaded status, and finally go to DELAY to try for another assignment. (If CKLAB found any recognizable label, it will have left the pack assigned to this control point; hence the necessity of calling MK40 to unload it here.)

If the pack name matches, we check D.FIRST; if this is zero, there was no visual identification number on the control card and we may skip to PRIVXE. Otherwise, D.FIRST ff. contain the number from the card, which must be checked with that in the pack label. We move the number from D.FIRST to D.FNT and then call subroutine PADVRN to copy it to VRNO ff., right justified with display code zero padding. Then we compare it with the number given on the pack, in LABEL+5 through LABEL+9. If they do not match, we follow the same procedure as above for non-matching pack name, except that the content of the dayfile message is different.

If the pack name and the visual identification number (if any) match, we arrive at PRIVXE; here we get the number of files on the pack from the 20th byte of the label, and call subroutine PAKFNT to set up an FNT entry for the pack. This has already been described above, for a new private pack. In the present case, subroutine AVOID will still do nothing if the FNT turns out to be full, because the pack is not in danger of having its label re-written until the pack FNT entry has actually been set up. If we exit from PAKFNT, however, the FNT entry has been set up, and we store the address of its third word in AVOIDA+1, so that from now on, if we fail to set up FNT entries for all tne files on the pack, because of a full FNT, the pack will be immediately released from the job and so its label will be saved from re-writing.

Note that PAKFNTA+1, in subroutine PAKFNT, contains the number of files on the pack, with which we entered PAKFNT; as the subroutine will not be called again, we can use this cell as a counter indicating when all the files have been dealt with.

We now enter a section of the program that is fairly complicated because it involves setting up RBT chains for all the files on the pack. D.FIRST through D.FIRST+3 are used in this way: D.FIRST points to the first byte of the pair of CM words last taken from the PRU area at LABEL; this is initialized so that the next word pair will have to come from the next PRU, as the list of files and their RBT chains begins in PRU 1 of record block 0. D.FIRST+1 contains the address of the FNT slot for the file currently being worked on. D.FIRST+2 contains constant 2, for use in moving word pairs of RBT chains. There is probably no reason why it could not have been set to 2 just above NXTFILE, once for the pack, instead of below PVK, once per file. D.FIRST+3 is filled with the first byte of the new word pair, to which D.FIRST points, by subroutine READON each time it is called to get the next word pair from the label.

The handling of each file from the existing pack begins at NXTFILE; we reduce the file count by 1, and exit from 5DA with 0 (indicating success) in the A register if this would make the count negative. Otherwise, we set PASTRBT to 0, indicating that we have to start a new RBT chain; then call subroutine READON to get the next word pair from the pack. If the exit is with zero in the A register, it means the first byte of the next word pair is zero, which would mean no more files on the pack and would in fact be an error. In this case we just exit from 5DA as if for success; at the end of the job, when the pack label is re-written, the file count will still be too high by the same amount. Otherwise, D.FIRST points to the start of the word pair, which should be the first two words of the FNT entry for the file. Calling subroutine SKFNT with the content of D.FIRST in the A register, we check for a file of the same name already at the control point. If the exit is with non-zero in the A register, there is such a file, and we exit from 5DA with a pointer to a duplicate-file message in the A register; 1AJ will issue the message to the dayfile. But before exiting, we call subroutine AVOID to release the pack from the job so that its label will not be re-written and spoiled by 1EJ.

If the exit from SKFNT is with 0 in the A register, we call SKFNT again, with 0 still in the A register, to look for an empty slot for the file FNT. If it finds none, it will call AVOID and then exit from 5DA with the FNT-full message pointer. If an empty slot is found, we return from SKFNT with its address in the A register, and store it in D.FIRST+1. D.FIRST points to the next word pair from the label, which should be the first two CM words of the FNT entry for the file, and we copy these into the first two words of the slot SKFNT found. Then drop pseudo-channel CH.FNT, which SKFNT left reserved in anticipation of this use. Note that the control point number in the fourth byte of the first word has already been altered to the present number by SKFNT, in order to assist in checking for an existing file of the same name at our control point; this was done on the last call but one to SKFNT. Next the second and third words of the new FNT entry will have to be re-written in CMR. If the second byte of the second word, as it came from the label (5+C.FFRBA,D.FIRST) is 0, the file is empty; if not, we go to PVFA. If empty, we zero D.Z1 through D.Z5, where the new form of the second word will be constructed; then copy the EST ordinal into the correct byte for an empty file on allocatable equipment, and call subroutine FINFNT to complete the second word by inserting the equipment code 0700B, store it in the FNT, and store VFD 60/1 as the third word of the FNT entry. Then return to NXTFILE to process the next file on the pack, if any.

At PVFA, we have to copy the RBT chain for the file from the label to the system RBT. First call subroutine READON to get the next word pair of the label, which will be the first RBT word pair for the file. Then reserve pseudo-channel CH.RBT, to entitle us to work on the RBT. Now if the pointer to the empty RBT chain, in CM cell P.RBT, is non-zero, there is a space we can use, and we go to PVG. If zero, we can do nothing until the RBT space has been extended, so we drop the pseudo-channel and call 1SX to a different PP, to get more RBT space. Then keep pausing and re-checking the empty chain pointer until it is non-zero, when we go back up to RESERVE and try again.

At PVG, we are to copy the next RBT word pair from the label to an available space in the RBT. PASTRBT contains 0 if this is the first word pair for the file, in which case we go to SETIT to link it with the FNT entry that has not yet been completed. We zero D.Z1 through D.Z5, where the FNT second word will be constructed, and copy the empty chain pointer into the second and third bytes of the word. From the second byte of the word pair in the label, we get the first RB byte number, and put this, along with 0 RBT ordinal, in the fourth byte of the word. The fifth byte of the word is left zero, for PRU 0. Then call subroutine FINFNT to complete the second word, and copy it and a third word = VFD 60/1, into the FNT. This brings uns to SETIT2.

If, at PVG, PASTRBT does not contain 0, this is not the first RBT word pair for the file and PASTRBT contains the pointer to the previous one in CMR. Instead of being linked to the FNT, the new RBT word pair must be linked to the previous one, in which the first (link byte) was left zero. So we alter the link byte in the preceding RBT pair from zero to point to the beginning of the empty chain (still where we read it from CM word P.RBT). Then branch to SETIT2.

Whichever way we come to SETIT2, we have now to store the next RBT word pair from the label into the first word pair of the empty chain. We save the pointer to the empty chain in PASTRBT, so that the next word pair if any can be linked to it. Then zero the first byte of the word pair as it came from the label; subroutine READON has already saved this byte at D.FIRST+3 where we shall look at it later; but at the moment the word pair is to be stored in the CMR RBT with no forward pointer. Bits 3-11 of the second byte of the word pair are now altered to the RBR ordinal of the disk pack; this was set by subroutine CKEST. Before using the first space in the RBT empty chain for this word pair, we must detach it from the empty chain. So now we read the empty chain word pair, take its forward pointer, which of course points to the second pair in the empty chain (or to zero if there are no more), and insert it in CM word P.RBT as the pointer to the first pair in the empty chain. Note that D.TO through D.T4 have not been disturbed since we got to PVG. Finally, put the word pair from the label into the space we just took from the empty chain, to which PASTRBT points, and drop pseudo-channel CH.RBT.

D.FIRST+3 still contains the first byte of the last word pair, as it came from the disk pack label. If it is 0, this is the last word pair of the RBT chain for the file, so we go to NXTFILE to look for the next file on the pack. Otherwise, go to PVFA to read the next word pair from the label and continue the RBT chain for this file. Note that if there are n word pairs in the RBT chain for a file, the binary values of the first bytes are successively 1, 2, 3, ... n -1, 0. If n = 1, the value in the first word pair is of course 0. But the only use made of these numbers in the label is to distinguish a non-last (non-zero value) word pair from the last one (zero value in the first byte).

For a control card of the type

REQUEST(fname,PK,pname)

we branched to PFILE from PRIV on finding 0002B as the second parameter in D.FIRST. The PFILE procedure is in the overlayable part of the program, since nothing in it requires reading or writing a pack label. All that has to be done is to insert in the FNT an entry for an empty file "fname" assigned to private pack "pname". The third parameter, in D.FNT ff., has to be a valid pack name. We alter the leftmost bit to 1, to coincide with the form it would have in the FNT, and then leave it for the moment. However, if the left bit was already 1, we exit from 5DA back to 1AJ with a pointer to the bad-pack-name message in the A register.

The first parameter, or file name, is in D.BA ff., where subroutine VFN would expect to find it. With a pointer to the appropriate failure message in the A register, call VFN to check the name for acceptability. If bad, VFN will exit from 5DA to 1AJ with the pointer in the A register. If good, we return from VFN and match the proposed name against the list of forbidden names in table NAMTAB. These names are not allowed to a file on a private pack because at job termination time they would normally cause a file to be retained in the output stack with some disposition code. But disposition codes are not allowed on private pack files, because the system would not be able to get at such a file after job termination anyway. If the name is a forbidden one, we again exit from 5DA to 1AJ with a message pointer in the A register. Otherwise, we arrive at PFILG and call subroutine SKFNT to seek an FNT entry for the pack named in D.FNT ff. If the exit from SKFNT is with 0 in the A register, none was found, and we exit from 5DA with a pointer to the no-such-private-pack message. If the FNT entry is found, we save its address in D.T7, and read the entry into D.FNT through D.EST+4 (leaving the new file name still in D.BA ff.) The count of files already on the pack is in bits 0-11 of the second word of the FNT entry, i.e. in D.FNT+9. For the moment we merely check this to see that adding 1 to it would not put it over 63; if so , exit from 5DA with a message pointer.

The limit of 63 files per private pack is set because we want to use a reasonably small pre-assigned area for the label; viz. 3 record blocks, or 3x5x64=960 CM words. 63 files would use up 126 words for their FNT entries in the label, and at worst 126 more for RBT word pairs that didn't actually account for any record blocks. This would leave 708 CM words to serve efficiently as RBT word pairs, at 8 record blocks per pair; enough to account for 2832 record blocks, which is 835 more than the available number on a private pack. So about 208 CM words will certainly go unused, and as the number of RBT word pairs used at 8 RB's per pair can't increase, another 50 files could be accommodated, giving a maximum limit of say 113. However, 63 seems like enough in practice.

If the increased number of files would not be excessive, we store zero in D.FNT+9, so that D.FNT+5 through D.EST+4, the so-altered second and third words of the pack FNT entry, can serve as the second and third words of the FNT entry for the new file. Then call subroutine SKFNT to see if there is already a file at the control point with the name given in D.BA ff. If so, the exit from SKFNT is with its address in the A register and we branch to PVK to exit from 5DA

with the pointer to the duplicate-file message in the A register. If the exit from SKFNT is with 0 in the A register, there was no duplicate and we call SKFNT again with 0 in the A register, to look for an empty FNT slot for the new file. If SKFNT cannot find any, it will take an error exit from 5DA. Subroutine AVOID will be called by SKFNT but will not do anything, as it is presumably all right to re-write the pack label normally if the job terminates merely because a new file could not be begun; the existing files are preserved as they stood at that moment.

If we exit from this second SKFNT call, the A register contains the address of the vacant FNT slot, and we immediately write D.BA through D.BA+4 and D.FNT+5 through D.FNT+14 into it as the new file entry. D.FNT+5 through D.FNT+14 are an altered copy of the same words from the pack entry, as explained above; D.BA through D.BA+4 contain the new file name from the control card, to which the control point number and type number (3 for local) were added by SKFNT when it was called previously, to help it compare the new name with existing FNT entries.

D.T7 still contains the address of the FNT entry for the pack; we now add 1 to bits 0-11 of the second word of this entry, stepping the count of files on the pack. Then drop pseudo-channel CH.FNT, which is always left reserved by subroutine SKFNT when it succeeds in finding an empty FNT slot. Then go to ZEREX to exit from 5DA back to 1AJ with 0 in the A register, indicating success.

When 5DA is entered from 1AJ with 4 in the A register, we branch to DROP, to remove a file from a private disk pack. This routine is in the overlayable part of the program, because no reading or writing of the pack label is necessary.

The control card format is REMOVE(fname) or REMOVE(fname,pname). The file name is in D.BA ff., and the pack name, if any, is in D.EST ff.; if no pack name is given on the card, D.EST contains zero. We call subroutine VFN to check the file name in D.BA for acceptability, with a pointer to a bad-file-name message in the A register. If VFN finds the name bad, it will exit from 5DA with the pointer to the message in the A register.

If we return from VFN, the file name has an acceptable format. With a pointer to the name in the A register, we next call subroutine SKFNT to look for an FNT entry for a file of that name at this control point. If the exit from SKFNT is with 0 in the A register, there is no such file to be dropped, so we exit from 5DA back to 1AJ with, in the A register, the pointer to the no-such-file message. Otherwise, on exit from SKFNT, the A register contains the address of the FNT entry, and we store it in D.FNT; then read the second word of the FNT entry, and check the equipment type; if this is not 07xxB, the file is not on a disk pack and we go to DROPM to exit from 5DA with a pointer to the not-disk-pack-file message in the A register. If assigned to disk pack, we take the initial RBT pointer and insert it in the release-chain stack request at DRQ. If this number is zero, the file is empty and we skip to DROPB. Otherwise, read the RBT word pair it points to, take the RBR ordinal from the second byte of this, and store it at RBRORD. (What is stored at RBRORD is always 10B times the RBR ordinal, with the three extra bits on the right not necessarily zero. But whenever RBRORD is used, these three bits are discarded anyway.)

Once RBRORD has been set, we can call subroutine GRBRAD to put the address of the first word of the RBR table in the A register. On return, we read the first two CM words of the RBR in such a way as to put the DST ordinal into bits 6-11 of D.T0 and the EST ordinal into D.T3. Note that if the file was empty, we jumped over this part of the code with the second word of the FNT entry in D.T0 through D.T4, so that the EST ordinal was in D.T3 in this case as well. Then we extract the DST ordinal and store it in the proper position in the drop-chain stack request.

At DROPB, whether the file was empty or not, we read its EST entry (the EST ordinal being at D.T3 whichever way we come to DROPB) and verify that the unit is private and at our control point. If not, exit from 5DA to 1AJ with the pointer to the not-on-private-pack message in the A register.

If the EST shows a private pack at our control point, we skip over DROPM and DROPE, and scan the FNT for the entry for this pack, identifying it by its having the leftmost bit of its first word = 1 (peculiar to pack pseudo-entries), being at our control point, and having the same EST ordinal as that for the file, in the fourth byte of its second word. If this FNT entryis not found, something is decidedly wrong, because we have identified the private pack fairly positively, and yet find no FNT entry for it. In this case we exit from 5DA with a pointer to the no-such-private pack message in the A register. If we do find the right FNT entry, we re-read its first word. Now if D.EST contains 0, there was no pack name in D.EST against that of the FNT entry. If there is no pack name on the card, or the right one, we arrive at DROPD; otherwise, we exit from 5DA to 1AJ with a pointer to the bad-pack-name message in the A register.

At DROPD, we check the initial RBT pointer for the file, which is in the release-chain stack request. If this is 0, the file was empty and we need not issue a request to evict it, so we jump to DROPL. Otherwise, evict the file by issuing the stack request at DRQ, and then arrive at DROPL. Note that the stack request is preset in the assembly of 5DA, all but the first RBT pointer and the DST ordinal, which we filled in above DROPB. At DROPL, we zero the file FNT entry, to which D.FNT points, and reduce by 1 the file count in bits 0-11 of the second word of the pack FNT entry, to which D.FNT+1 points. Then go to ZEREX to exit from 5DA with 0 in the A register, indicating success. However, if the file count in the pack FNT entry was already 0, we do not alter it, but exit from 5DA with a pointer to the file-count-already-zero message in the A register.

SUBROUTINES

MK40

MK40 is called to alter the EST status of the disk pack we have been working on from 402pB, where p is our control point number, to 4040B, the unloaded status. Subroutine MEST is called twice, to alter the status to 4020B and then to 4040B. Probably the intermediate step 4020B is unnecessary. On each return from MEST, the A register contains 0 if and only if the requested change was granted by monitor. We need not bother checking the request for 4020B, but we check the final request for 4040B, and if it is not granted, we call subroutine ERMES to issue the dayfile message at CANTUNL and then abort the job. If the final request is granted, we exit from MK40.

Other Subroutines Called

MEST, ERMES.

Registers Destroyed

None directly by MK40.

Entry and Exit Information

None.

PAKFNT

This is called to put a pseudo-entry for the disk pack into the FNT, and to store its RBR table in CMR. On entry, the A register contains the file count for the pack, and this is saved. Then subroutine STORBR is called to copy the RBR table, which has been read into or constructed in LABEL+40 through LABEL+239, into the position in CMR determined by RBRORD. Note thatthis RBR table remains there even if the FNT entry cannot be set up, as 5DA will then abandon the disk pack in unloaded status, and the exact content of the RBR table in CMR will not matter. It will be correctly re-written whenever the disk pack is brought into an active status in the future.

With 0 in the A register, we call subroutine SKFNT to find an empty FNT slot. If it fails, SKFNT will exit from 5DA. If it succeeds, we return from SKFNT with the FNT address in the A register, and use this immediately to write the first word of the pack entry into the FNT. This first word is in D.BA through D.BA+4; the name itself was put in D.BA through D.BA+3 by 1AJ before calling 5DA, and D.BA+4 was left zero; 5DA has already set the leftmost bit of D.BA to 1, as appropriate for a pack pseudo-entry in the FNT, and has already called SKFNT once before to see if there was already such an entry with this name; SKFNT on that occasion added the file type number (local=3) and the control point number into D.BA+3 to simplify the search.

On the present occasion, SKFNT has left D.Z1 through D.Z5 all zero, and the address of the first word of the FNT slot in D.Z6. Now we modify D.Z1

through D.Z5 into the second word of the FNT entry by putting the equipment type into the first byte, the EST ordinal (previously found by subroutine CKEST and stored at ESTO) in the fourth byte, and the file count into the fifth byte. Then we write this into the second word position of the FNT slot; and write VFD 60/1 into the third word position. Finally, we release pseudo-channel CH.FNT, which SKFNT had left reserved for us after finding an empty slot, and then exit from PAKFNT.

Subroutines Called

STORBR, SKFNT, R.DCH

Registers Destroyed

None by PAKFNT directly.

Entry Information

The file count for the pack, in the A register.

Exit Information

None.

STORBR

This is called to copy the RBR table for the disk pack, which has been read into or constructed in LABEL+40 through LABEL+239, into the proper position in the CMR as determined by RBRORD. First we copy the correct DST ordinal and unit number into the model RBR; they were read from the same RBR table in CMR into RBRHEAD by subroutine CKEST. Then we copy the EST ordinal into the model RBR; this was stored at ESTO by CKEST. Then call subroutine GRBRAD to put the address of the RBR table in CMR into the A register; it uses RBRORD, which was set by CKEST. On return from GRBRAD, write the RBR table into CMR and then exit from STORBR. Note that D.SV2 is initially set by 5DA to 38, the number of CM words in an RBR table, and this value is never altered in 5DA.

Subroutines Called

GRBRAD.

Registers Destroyed

None by STORBR directly.

Entry and Exit Information

None.

VFN

This is called to check that the file or pack name in D.BA through D.BA+4 has a plausible format. On entry, the A register contains the address of the dayfile message to be issued if not, and this is saved. Then we check that the first character is a letter; that the 8th through 10th characters are 00B; that the other characters are either 00B or letters or digits, and that there is no 00B character to the left of the last non-zero character. If so, we merely exit from VFN. If not, we pick up the address of the error message and go to FIVEDX to return to 1AJ with this address in the A register.

Subroutines Called

None.

Registers Destroyed

D.Z1, D.Z2

Entry and Exit Information

Only the error message pointer in the A register, in case of failure.

PADVRN

This is called to format the visual pack identification "number" (though it may consist of letters and/or digits indifferently). On entry, the ident. is left justified with binary zero fill in D.FNT through D.FNT+2, essentially as read from the control point area. We first scan this area from the right, find the number of non-zero characters, and put it in D.Z2. If the number is even, skip to VRNG; if odd, shift the field one character rightward and insert a display code zero on the left, to make up an even number. At VRNG, we then insert six display code zeroes as possible fillers to the left of the field. D.Z2 has been adjusted to contain half the number of characters, after the number was adjusted up from odd to even if necessary; i.e., the number of bytes of ident. from D.FNT to the right end. Adding D.T5 to this number gives us the start of a three-byte field containing what we got from the control point area, with enough display code

zeroes on the left to make up six characters. We move these three bytes to VRNO ff., where they are available for comparison purposes, and also form the last six characters before the right parenthesis in the dayfile message beginning at VRMSG. Then we exit from PADVRN.

Subroutines Called

None.

Registers Destroyed

D.Z1, D.Z2, D.T5, D.T6, D.T7

Entry and Exit Information

None.

PAUSE

This is called at only one point, when we are waiting for 1SX in another PP to get some new RBT space. It could have been called while we are waiting for the operator to type in a pack EST assignment and a visual identification number, but instead we recall 1AJ (and hence 5DA) with a 5-second delay. PAUSE calls resident subroutine R.PAUSE, and on exit updates D.FL and D.RA. Then it exits if the error flag from the control point is 0, or else goes to ABORT to abort the job.

Subroutines Called

R.PAUSE

Registers Destroyed

D.T0 through D.T4, by R.PAUSE.

Entry and Exit Information

None, except that D.RA and D.FL are updated.

MEST

This is called with a number in the A register which, added to 4000B, is to be the new EST status for the disk pack. We set up the proper M.ESTZ monitor request and issue it through R.MTR, and then compare the new status, in D.T2, and with what was requested (D.T2 will actually contain the new status minus 4000B). We exit with the A

register containing the result, which will be 0 if and only if the requested status was granted.

Subroutine Called

R.MTR

Registers Destroyed

D.T0 through D.T4

Entry Information

The desired EST status, minus 4000B, in the A register. The EST ordinal of the pack in ESTO.

Exit Information

0 in the A register if and only if the desired status was obtained.

GET

This is called to bring the disk pack whose EST ordinal is in ESTO to our control point, i.e., to alter its EST status to 402pB, where p is the control point number. We put 2pB in the A register and call subroutine MEST. If the return is with 0 in the A register, exit from GET without changing it, to indicate success. Otherwise, we have not obtained the disk pack, and call subroutine ERMES to issue the dayfile message CANT GET PACK ASSIGNED TO C.P. Then we check D.TR; this will contain 3 if and only if 5DA was called by 1AJ for an RPACK or REQUEST control card. (REQUEST, however, never causes a call of GET, as the pack need not be referenced.) If so, exit from GET with non-zero in the A-register, to indicate failure. If not, 5DA was called by an operator type-in of DEVADD, UNLOAD, or BLANK, and we merely abort the job.

Subroutines Called

MEST, ERMES

Registers Destroyed

None by GET directly.

Entry Information

None.

Exit Information

0 in the A register if successful; non-zero if unsuccessful in case of RPACK.

WRITEP

This is called to write a PRU on a disk pack, taken from LABEL through LABEL+477B. The stack request at ZERQ is used, and it must already have been completed except for the order code in the fourth byte. In fact, WRITEP is called only to write PRU number zero, record block zero, for which the stack request at ZERQ is pre-assembled; so all that needs to be filled in before calling WRITEP is the EST ordinal, at EST0, and the DST ordinal, at DST0

Subroutine Called

R.WRITEP

Registers Destroyed

D.T0 through D.T7, by R.WRITEP.

Entry and Exit Information

None.

READP

This is called to read a PRU from a disk pack into LABEL through LABEL+477B. The stack request at ZERQ is used, and it must have already been completed except for the order code in the fourth byte. READP is mostly called to read PRU number zero, record block zero, for which the stack request at ZERQ is pre-assembled; so all that needs to be filled in before calling READP is the EST ordinal, at EST0, and the DST ordinal, at DST0. Subroutine READON is the exception; whenever it calls READP, this is to read the next PRU; however, before READON is called for the first time, PRU 0 has been read/

Subroutine Called

R.READP

Registers Destroyed

D.T0 through D.T7, by R.READP

Entry and Exit Information

None.

CKLAB

This is called to get the label PRU of the disk pack whose EST ordinal is in ESTO, and check that it contains a recognizable label. If so, we exit from CKLAB with 0 in the A register, and if not, with non-zero.

First we call subroutine GET to bring the disk pack to our control point; if it succeeds, the exit is with zero in the A register and we continue with CKLAB; if not, GET will either abort or (depending what sort of control card is being processed) exit with non-zero in the A-register; if the latter, we exit from CKLAB with non-zero in the A register.

If there is a good return from GET, we call subroutine READP to read the first PRU of the disk pack into LABEL through LABEL+477B; then subroutine CKSUM, which forms the checksum of every byte in this area except LABEL+39, and exits with the sum in the A register. Subtracting this sum from LABEL+39, the checksum byte, checks the checksum. If good, we continue. If not, we go to CKLABY, where we find out from D.TR whether we are processing a BLANK type-in. If not call subroutine ERMES to issue the bad-label dayfile message; then subroutine MK40 to release the pack from our control point; and then exit from CKLAB with non-zero in the A register. But in case of a BLANK type-in, CKLAB can only have been called from CZLAB, and an unrecognizable label would be just as good as a blank label for a BLANK. So in this case we take the exit from CZLAB, with 0 in the A register, indicating that the label PRU is eligible to be over-written with a blank label.

If the checksum is good, we skip over CKLABS, and see whether the label begins with "DEV1". If not, go to CKLABY as for a bad checksum, and if so, continue. Next see that characters 6-10 ofthe label are all decimal digits, giving a Julian date YYDDD in which YY is between 69 and 99 and DDD is between 001 and 366. If so, exit from CKLAB with 0 in the A register, indicating that the pack appears to have a recognizable label; if not, go to CKLABY as for a bad checksum.

Subroutines Called

GET, READP, CKSUM, ERMES, MK40

Registers Destroyed

D.Z1 through D.T0 by CKLAB itself.

Entry Information

None.

Exit Information

0 in the A register indicating a recognizable label, or non-zero for unrecognizable. When processing a BLANK type-in, however, an unrecognizable label will cause an exit from subroutine CZLAB, which must have called CKLAB.

CZLAB

This is called to verify that the disk pack whose EST ordinal is in EST0 has a blank label (or, in the case of a BLANK type-in, no recognizable label). We call subroutine CKLAB to get the pack, read its first PRU, and see if this shows a recognizable label. If so, the return from CKLAB is with 0 in the A register; if not, with non-zero. However, if we are processing a BLANK type-in and there is no recognizable label, we will have exited from CZLAB with 0 in the A register, without formally exiting from CKLAB. If the return from CKLAB is with non-zero in the A register, we exit from CZLAB with non-zero. If with zero, we check the pack name field to see if it contains binary zero, showing a blank label. If so, exit from CZLAB with zero in the A register, indicating success. If not, and we are processing a BLANK type-in, exit from CZLAB with non-zero in the A register. This will make 5DA exit back to 1BT with the address of the label in the A register, so that 1BT can format a message asking the operator whether to blank label this privately-labeled disk pack or not.

If CZLAB finds a non-zero pack name field and we are not processing a BLANK type-in, we go to CKLABZ with a pointer to the already-labeled message in the A register. At CKLABZ, subroutine ERMES is called to issue the message; then subroutine MK40 to release the disk pack from our control point. Then, with non-zero in the A register, we exit from subroutine CKLAB; but as we did not enter it properly just now, the exit is governed by the last proper entry, which was at the instruction next below location CZLAB. So this exit from CKLAB brings us to the NJN CZLABX just below CZLAB, and having non-zero in the A register, we exit from CZLAB. This roundabout procedure mereley saves three instructions at the end of subroutine CZLAB.

Subroutine Called

CKLAB

Registers Destroyed

None by CZLAB itself.

Entry Information

The EST and DST ordinals of the disk pack, in ESTO and DSTO.

Exit Information

Zero in the A register if the pack has a blank label, or, in the case of a BLANK type-in, has an unrecognizable label. Otherwise non-zero.

CKEST

This is called, with an EST ordinal in ESTO, to verify that the ordinal refers to a disk pack, to find its RBR entry, read the first two CM words of the RBR entry into RBRHEAD ff., and to store its RBR ordinal and DST ordinal at RBRORD and DSTO.

First we set a constant 2, for use lower down in reading the first two words of the RBR from CM. Then get the EST pointers and see that the EST ordinal is not greater than the length of the EST. If it is, call subroutine ERMES to issue the range-error message, and then exit from CKEST with non-zero in the A register. Next read the EST entry, and see that it refers to a disk pack (code AP); if not, call ERMES to issue the non-pack message and exit from CKEST with non-zero in the A register.

Now we get y, the starting address of the request stack in CM, from cell P.RQS, and save it in D.Z6 and D.Z7; and x, the starting address of the first RBR table, and store it in D.T0 and D.T1. y is either the last+1 address of the last RBR table, or is up to 37 greater than that. In other words, each RBR table occupies 38 words; if there are n RBR's, y=x+38n+k, where k is between 0 and 37. We also set RBRORD, in effect, to 8(-1), as we shall add 1 to it before testing each RBR, and we want it ultimately to contain 8 times the RBR ordinal.

At SCC we read the first two words of what may be the next RBR table in CM, and then add 38 to the pointer. Also add 10B to RBRORD, to give 8 times the RBR ordinal of the table we are now to look at, if it is one. But if the

RBR table pointer {last+1 address of the RBR table we are about to look at, if it is one} is greater than the address of the start of the request stack, this is not an RBR table, and we have not found the match we were seeking. So we issue the no-RBR dayfile message and exit from CKEST with non-zero in the A register. Otherwise, we do have another RBR table to look at, and we compare the EST ordinal in its seventh byte with ESTO. If they do not match, return to SCC and try for another RBR table. If they do match, set DSTO to contain the DST ordinal, from the second byte of the RBR table. RBRORD is already set. Then exit from CKEST with 0 in the A register.

Subroutine Called

ERMES

Registers Destroyed

D.Z1 through D.T5, D.FNT+5.

Entry Information

The disk pack EST ordinal in ESTO.

Exit Information

The DST ordinal in DSTO, the RBR ordinal times 8 in RBRORD, the EST entry in D.EST through D.EST+4, the first two CM words of the RBR table in RBRHEAD through RBRHEAD+9.

CKSUM

This is called to find, and leave in the A-register, the 12-bit sum, with end-around carry at each addition, of bytes LABEL through LABEL+477B, excluding LABEL+39. LABEL+39 is the checksum byte in the first 500B-byte PRU of a disk pack, so we have to sum all the other bytes either to find a new checksum or to check an old one. First we save the content of LABEL+39 in D.Z3; then zero LABEL+39; then accumulate the checksum of LABEL through LABEL+477B in D.Z2; then restore LABEL+39; and finally exit with the content of D.Z2 in the A register.

Subroutines Called

None.

Registers Destroyed

D.Z1, D.Z2, D.Z3

Entry Information

None.

Exit Information

The checksum in the A register.

GRBRAD

This subroutine takes RBRORD to have been set by subroutine CKEST to 10B times the RBR ordinal of the disk pack. It calculates the address of the beginning of the corresponding RBR table in CM, and exits with this address in the A register. The starting address of RBR table number 0 is taken from bits 36-59 of CM word P.RBR.

Subroutines Called

None.

Registers Destroyed

D.Z1 through D.Z5

Entry Information

RBRORD properly set.

Exit Information

The starting address of the RBR table in the A register.

FINFNT

This writes the second and third words of the FNT entry for a private pack or a file assigned to a private pack into the proper position in the FNT. D.FIRST+1 contains the address of the first word of the FNT slot. D.Z1 through D.Z5 have already been set up as the proper second word of the first byte. The third word is set up merely as VFD 60/1.

Subroutines Called

None.

Registers Destroyed

D.Z1 through D.Z5

Entry Information

The FNT pointer in D.FIRST+1, and the partly-constructed second word of the FNT entry in D.Z1 through D.Z5.

Exit Information

D.FIRST+1 has had 2 added to it.

READON

This is called to fetch the next word pair (10 PP words) from the label of the current disk pack. The pointer to the preceding word pair is in D.FIRST. We add 10 to it, and if it now equals LABEL+500B, we have exhausted this PRU and must begin the next. Otherwise, we have the pointer to the new word pair in D.FIRST; we go to READONY, copy the first byte of the word pair into D.FIRST+3, and exit from READON with the first byte in the A register.

As this is the only place in 5DA where we read or write any PRU other than the first one on the pack, all we have to do to address the next PRU is to increase the PRU number in the stack request at ZERQ by 1. If the PRU number thus becomes 5, we reset it to zero and add 1 to the record block number in the stack request. Then call subroutine READP to issue the stack request. On return from READP, we set the pointer in D.FIRST+1 to LABEL, the beginning of the first word pair in the new PRU; copy the first byte of the word pair into D.FIRST+3, and exit from READON with the first byte in the A register.

Subroutine Called

READP

Registers Destroyed

D.T0 through D.T7, if READP is called.

Entry Information

The stack request at ZERQ correctly set to read the PRU now in LABEL through LABEL+477B. The pointer to the last word pair used, in D.FIRST+1. This pointer is actually set to LABEL+500B-10 before calling READON the first time, to force it to begin with the first word pair in the second PRU.

Exit Information

The pointer in D.FIRST+1 is updated. The new PRU, if necessary, is in LABEL through LABEL+477B, and the stack

request is set for the PRU most recently read. The first byte of the new word pair is in D.FIRST+3 and in the A register.

SKFNT

This is called with a pointer in the A register, to scan the FNT either for a vacant slot, if the pointer is 0, or for an entry at our control point with a name the same as the seven characters beginning in the cell to which the pointer points. On exit, the A register contains 0 if what was sought was not found, or else a pointer to the matching entry or empty slot in the FNT.

First we store the pointer in D.Z1. If it is zero, calling for an empty slot, we branch to SKFNTA and read a zero CM word into D.Z0 through D.Z4. This is what we will match successive FNT slots against. Then reserve pseudo-channel CH.FNT, as otherwise an empty slot we found might not stay empty. This brings us to SKFNTB.

If the pointer is non-zero, it points to a potential file name in PP memory. We alter the fourth and fifth bytes of the potential name area to coincide with the form of the first word of an FNT entry for a file of this name at our control point, type local. Then branch to SKFNTB.

At SKFNTB, we have D.Z1 pointing tothe first of five PP cells that contain the model of the FNT entry or vacancy we are seeking. The only mis-match allowed will be in the type field of the word, bits 15-17. Now we get the FNT pointers and scan the FNT. If we find a match, we exit from SKFNT with the pointer to the FNT entry or slot in D.Z6 and in the A register. If no match, and if D.Z1 is not zero, we exit from SKFNT with 0 in the A register, having failed to find a matching FNT entry. But if no match, and D.Z1 contains 0, we have failed to find an empty slot; the FNT is full, and 5DA is not going to be able to complete its work. So we drop pseudo-channel CH.FNT, as we are not going to do anything to the FNT; call subroutine AVOID to save the disk pack label from destruction if necessary; then branch to FIVEDX and back from 5DA to 1AJ with a pointer to the FNT-full message in the A register.

Subroutines Called

R.RCH, R.DCH, AVOID

Registers Destroyed

D.Z0 through D.T4

Entry Information

In the A register, 0 if an empty FNT slot is to be sought; otherwise a pointer to the first of five bytes whose first seven characters contain a name whose like is to be sought in an FNT entry at our control point.

Exit Information

If the entry was with 0 in the A register, then if we find a vacant FNT slot, we exit with a pointer to it in the A register and in D.Z6, and with pseudo-channel CH.FNT reserved. If the FNT was full, CH.FNT is not reserved; the disk pack has been dropped from our control point if necessary, and 5DA returns to 1AJ with a pointer to the FNT-full message in the A register.

If the entry was witn non-zero in the A register, the 5 bytes to whose first one it points have been adjusted to look like the first word of an FNT entry for a file of that name, type local, at our control point. If no match to it was found in the FNT, the A register contains 0 on exit; if a match, the FNT address is in D.Z6 and in the A register.

AVOID

This is called to drop the disk pack from our control point, if necessary, when subroutine SKFNT finds the FNT is full, or when, in processing a control card of the form RPACK{pname,E} or RPACK{pname,E,vrno} we find one of the files on the pack has the same name as an existing file at our control point.

If AVOIDA+1 is 0, we exit from AVOIDA without doing anything. If non-zero, it has been set so during the processing of an RPACK{pname,E} or RPACK{panme,E,vrno} control card, just above loaction NXTFILE, to contain the address of the third word of the FNT entry for the private pack. Once the private pack entry has been set up in the FNT, if we simply abandon the process of inserting all the files on the pack into the FNT, we will have the pack label re-written at job termination with one or more of the pack's files missing. As the pack should not suffer this loss, even if it cannot be used by the job at this time, AVOIDA+1 has been set so that when AVOID is called, we write zero into the FNT slot for the pack FNT entry, and call subroutine MK40 to unload the pack from our control point. Then at job termination, the pack label will be left unchanged.

Subroutine Called

MK40

Registers Destroyed

D.Z1 through D.Z5

Entry Information

In AVOIDA+1, either zero {for do nothing} or the address of the third word of the FNT entry for the private pack.

Exit Information

None.

LOD7DA

5DA is rather cramped for space; the error messages are numerous and bulky; but once an error message is needed, whatever may have been read from the disk pack into LABEL through LABEL+477B is no longer needed. {The only exception is when 1BT calls 5DA to blank label a pack, and 5DA finds a private label on the pack already. In this case 5DA keeps LABEL through LABEL+477B intact and passes address LABEL back to 1BT, and 1BT has the error message in its own field length.} So we have all the error messages for 5DA gathered into PP overlay 7DA, which is assembled as a segment of 5DA so that its addresses are available to 5DA through the assembly.

LOD7DA is entered with the address of some message from 7DA in the A register; this is saved; then 7DA is loaded into the space beginning at 7200B=LABEL; and then we exit from LOD7DA with the address of the message again in the A register--only now the message is in PP memory at that address, which it was not before

LOD7DA is called by subroutine ERMES {see below} when 5DA wants to issue an error message and then either continue its work or abort the control point immediately. When 5DA is to return control to 1AJ, with the message pointer in the A register, it branches to FIVEDX with the pointer; LOD7DA is called there so that the pointer will not be meaningless, and then we take the subroutine exit from 5DA with the pointer again in the A register.

Subroutine Called

R.OVL

Registers Destroyted

D.Z1 through D.T7 by R.OVL

Entry and Exit Information

The pointer to the message, in the A register. Only on exit is the message actually in memory.

ERMES

This is called whenever 5DA wants to issue an error message without returning control to 1BT or 1AJ. We enter with the address of the message in the A register; call subroutine LOD7DA to load the messages, in overlay 7DA; call R.DFM to issue the message to the dayfile, and exit from ERMES.

Subroutines Called

LOD7DA, R.DFM

Registers Destroyed

D.Z1 through D.T7, LOD7DA and R.DFM

Entry Information

The address of the message, in the A register.

Exit Information

None.

6PC--DROP PERMANENT MASS STORAGE FILES

Preset

D.PPMES 1 = PP message buffer first word address
D.FA = FST word one address
D.CPAD = CP No. *200B

Temporary

D.Z1 - D.T7

Set

D.RA and D.FL by Pause for Relocation

Routing Area

C.PP6WA - L.PPHDR 7777B

Error Messages

Headed by ▽PFM SYSTEM ERROR▽;

1. ▽ORBTC NOT IN FNT▽.
   for ORBTC not found as a file name in FNT

2. ▽BAD PFD POINTER▽.
   for incorrect PFD pointer in APF entry

Following messages will have Permanent File Name, followed by:

3. ▽CYCLE NOT FOUND▽.
   for the cycle specified in APF not found in PFD

4. ▽CYCLE DUMPED▽.
   for the cycle specified in APF not available in PFD

5. ▽NAMES DISAGREE▽.
   for names in PFD and in RBTC disagreeing

6. ▽CYCLES DISAGREE▽.
   for cycles in PFD and in RBTC disagreeing

7. ▽RBT - RBTC DISAGREE▽.
   for file RBT chain disagreeing to RBTC up to the end of RBTC

{For System Errors, File RBT chain will be released without releasing RBR bits.}

SPC--DROP PERMANENT MASS STORAGE FILES

Preset

D.PRMES 1 = PP message buffer first word address
D.FA = FST word one address
D.CPAD = CP No. *2005

Temporary

D.Z1 - D.T7

Set

D.RA and D.[illegible] by Pause for Relocation

[illegible] Area

C.[illegible] - L.PPHDR [illegible]

Error Messages

Headed by "PPN SYSTEM [illegible]"

1. "ORBTC NOT IN FNT"
   For ORBTC not found as a file name in FNT

2. "BAD PFD POINTER".
   For incorrect PFD [illegible] in APF [illegible]

Following messages will have Permanent File Names followed by

3. "CYCLE NOT FOUND".
   For the cycle specified in APF not found in PFD

4. "CYCLE DUMPED".
   For the cycle specified in APF not available in PFD

5. "NAMES DISAGREE".
   For names in PFD and in ORBTC disagreeing

6. "CYCLES DISAGREE".
   For cycles in PFD and in RBTC disagreeing

7. "RBT - RBTC DISAGREE".
   For file RBT chain [illegible] agreeing to RBTC to be the end of RBT.

(For System Errors, File RBT chain will be released without releasing RBR bits.)

MSG

General

MSG is a PP program, always loaded at 1000B. It is called by a CPU program in order to get a message added to the dayfile. A peculiarity of the way it is called is that if without recall, the call request points directly to the message; while if with recall, the request points to a word whose left half points to the message itself, while its right half is zero, in order that the rightmost bit of the word the call points to shall be certainly 0, and available for setting to 1 when the request has been fulfilled. The macro generator for the ▽MESSAGE▽ macro does not take account of this; it generates an RJ CPC followed by a pointer to the message, and CPC, when entered, provides the necessary intermediate word if the request is indirect.

Entry Information

The input register initially contains:

```
        VFD       18/3RMSG,2/a,4/c,12/d,6/e,18/f
```

where:
a is 1 if recall or 0 if no recall,

c is the control point number,

d is 0 if the message is to be copied to the control point B-display and written on the control point dayfile and possibly the system dayfile; or non-zero if the message is only to be copied to the B-display.

e is 2 if the message is a ▽hardware error file▽ message. It will only be written on the h.e.f.; not to the control point or system dayfile or the B-display; and the value of d in the request will be disregarded.

e is 1 or 0 according as the message is not to be written, or is to be written, to the system dayfile. In either case, it will also be copied to the B-display. However, if d is non-zero, the message will not be written to either dayfile.

Strictly speaking, the case for e=2 applies for any value of e in which its bit 1 is 1; i.e., for e=2,3,6,7,......76,77; and the case for e=1 applies for all other non-zero values of e.

f is the address, relative to RA, of the first word of the message if a=0. If a=1, f is the address of a word having the form 30/g, 30/0, g being the address relative to RA of the first word of the message. In the latter case, bit 0 of the word at RA+f will be set to 1 when the message has been issued.

Entry Information {continued}

A ▽hardware error file▽ message is always 6 CM words long. The length of any other message is defined by its ending with the first zero byte that falls in bits 0 - 11 of a CM word. Such a message must not be more than 80 characters long. If it is more than 40 characters long, the first 40 characters will be written to the dayfile{s} as appropriate, and the remainder will be written to it or them as a second message. The whole length, up to 80 characters, will be written to the B-display area of the control point; if the message is 50 or fewer characters long, this will create a new ▽first message line▽. If more than 50 characters long, the first 50 characters will become the ▽first message line▽, and the remainder will become the ▽second message line▽.

Exit Information

None, if the request was without recall. If with recall, bit 0 of the word to which the request pointed is set to 1.

Other Programs Called

Monitor requests are made in the normal way, but otherwise no other programs are called.

Narrative

We begin by setting ON {D.PPONE} to contain 1, and reading the input register into IRB through IRB+4 {D.PPIRB through D.PPIRB+4}. Then copy the address from the request into ARG and ARG+1, and with this in the A register call R.TFL to add RA to it. If the return is with the A register negative, the address is out of range, and we go to MSG5 to issue the dayfile message ▽MSG - ARG ERROR▽ and abort the control point. Otherwise, read the word the request points to into TMP through TMP+4. Next extract the recall bit from the input register image and store it in RCL. If zero, branch to MSG1; ARG and ARG+1 point directly to the message, but we shall not make use of the fact that its first CM word has already been read into TMP ff. If non-zero, we extract the address of the message from bits 30-46 of the CM word read into TMP ff., and store it in ARG and ARG+1.

At MSG1, in either case, we have the address of the message in ARG and ARG+1. Now if bit 19 of the input register image is 0, we go to MSGP to process a normal message. If 1, we have a hardware error file message, and set D.T1 to contain 20B as an indication to monitor of this. For an H.E.F. message we do not call PP resident subroutine R.DFM, which would set D.T1 to zero before calling R.MTR with M.DFM in the A register; we will call R.MTR directly from MSG; so we set D.T1 to 20B as part of the latter call. Then set WC to contain 6, the known length of the message, and D.T2 to contain the address of the sixth word of our PP's message buffer; this is also part of the call we shall make to R.MTR. Then pick up the address of the message in the user's field length, add 6 for the length of the message, and call R.TFL to add RA to it. If the return is with the A register negative, the message address was out of range and we go to MSG5

Narrative {continued}

to issue the dayfile message ▽MSG - ARG ERROR▽ and abort the control point. Otherwise, subtract 6 from the A register to get the initial address of the message, and read the message into BUF ff.; then write it into the first 6 words of our PP's message buffer; call R.MTR to issue the dayfile message; and go to MSG10 to set the completion bit if the request was with recall, and drop the PP.

We come to MSGP if the message is not for the H.E.F., and set WC, the word counter, to zero. In the loop from MSGL to just before MSG7, we pick up the address in ARG and ARG+1, call R.TFL to add RA to it, and branch to MSG5 if R.TFL answers that the address exceeds the field length; in that case we issue the dayfile message ▽MSG - ARG ERROR▽ and abort the control point. If the address is good, we read the word into the next available 5 bytes of the message buffer beginning at BUF; check each character for being less than 60B {characters above 57B could be printed in the dayfile, but might cause false coordinates if they were displayed on the A or B display of the console}; and replace any of the first four bytes by a pair of blanks if the byte is zero. If a character is not less than 60B, we go immediately to MSG7 to issue the dayfile message ▽MESSAGE FORMAT ERROR▽ and abort the control point.

At the end of checking each CM word of the message, we add 1 to WC, and then return to MSGL for the next word unless WC contains 8, the maximum length for a message, or the last byte of the last word read contained zero, the message terminator. In either of the latter cases, go to MSGE.

We come to MSGE with the message in BUF ff., and its length in CM words in WC. Now we extract the control point number from the input buffer image, and copy the whole message into the B-display area of the control point. Next, if bits 24-35 of the input register contained zero, we go to MSG13 to send the message to the dayfile{s}. If non-zero, we pass directly to MSG10, to set the completion bit if necessary and drop the PP.

At MSG13, we prepare for the possibility that the message is over 40 characters long; i.e., extends beyond BUF+19, by saving BUF+20 at IRB {which merely contained the leftmost two characters of the input register image, ▽MS▽}, and zeroing BUF+20 to provide a terminator for the first part if the message, if it is long. Between MSGL and MSG6, above, we maintained BP as a pointer to the last {not last+1} byte of the message assembled in BUF ff.; so now we use it to zero the byte after the last byte in the message. {This is in case the message ended without a zero byte; but it could only be so if the message were terminated because it extended through 8 CM words; hence we could just as usefully zero BUF+40, as 1,BP.}

Narrative (continued)

Now if bits 18-23 of the input register image are non-zero, we assume we have the case e=1, as explained above under "Input Information"; the message is not to go to the system dayfile, so we set to 1 bit 14 of the address that will be in the A register when we call R.DFM at SCM1 and SCM2. Otherwise, this bit remains 0.

At SCM1, we call R.DFM with the address of the beginning of the message in bits 0-11 of the A register; bit 12=1 to prevent R.DFM from copying it to the B-display, which MSG has already done, and bit 14 = 0 or 1 as explained in the preceding paragraph. Now if WC contains less than 5, the message was not over 39 characters, and we have sent it all and can go to MSG14. If 5 or more, the message was 40 or more characters. Byte BUF+20, which must have contained characters 41 and 42, has been saved at IRB; if this is zero, the message was exactly 40 characters followed by a zero byte, and as we have already sent the 40 characters we can go to MSG14, (however, in the code between MSG4 and MSG5, we altered to blanks any zero byte that did not come at the end of a CM word, so byte BUF+20 can hardly contain zero if the message was 5 or more CM words long.) Otherwise, we restore byte BUF+20. A zero byte has already been stored after the last byte of the message (just after MSG13 above); and we now call R.DFM with the address of the second part of the message in bits 0-11 of the A register, and bits 12 and 14 set as explained above for the first part of the message. Then arrive at MSG14.

At MSG14 is a piece of conditionally-assembled code. If IP.MSCT =0, it is skipped, and we go straight to MSG10. Otherwise, we subtract 1 from the byte in the control point area that contains the message limit count. If it is not reduced to zero, go to MSG10. If it is reduced to zero, issue the dayfile message "MESSAGE LIMIT EXCEEDED" and abort the job.

At MSG10, we check RCL, in which the recall bit of the input register image was initially stored. If this is zero, go to MSG11 to drop the PP. If not, set to 1 bit 0 of the word in the job's field length to which bits 0-16 of the input register image point; then go to MSG11 and drop the PP. This CM word was initially read into TMP through TMP+4, and is still there. R.TFL is used to add RA to its address, but we neglect the case of a return from R.TFL with the A register negative because it has already been called to add RA to this address, and aborted then if the address was beyond the field length; since then, the field length cannot have changed while MSG was active at the control point because it has never called R.PAUSE.

14.0 SYSTEM CENTRAL MEMORY ROUTINES

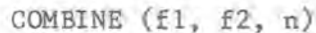
14.1 BKSP

Description

1. The backspace count specified in RA+3 is converted to octal.
2. An open alter (OPE function code = 120B) is performed on the file.
3. A SKIPB (CIO function code = 640B) is performed on the file.
4. An END request terminates processing.

14.2 COMBINE

This is a utility routine which operates in the CP. It is called with the following control card:



In this operation, n (decimal) logical records are read from file f1 and written as one logical record onto file f2, level zero. The operation proceeds as follows:

1. A five word FET is generated with

   Program length + 2 → FIRST = FET + 1

   Program length + 2 → IN = FET + 2

   Program length + 2 → OUT = FET + 3

   Field length - 4 → LIMIT = FET + 4

2. The number n is converted from decimal to binary.

3. An open request is made on the input file f1 to determine device type. The device type is indicative of the PRUSIZE. The device type field is in bits 48-59 of FET + 1. The PRU size will be recorded at the CM location PRUSIZE.

   - If the device type ≤ 12B, the device = mass storage, and 64 → PRUSIZE.

   - If the device type = 40B, the device = 1/2" tape, and 128 → PRUSIZE.

   Otherwise, assume device type = card reader, and 16 → PRUSIZE.

4. READ's from f1 are issued as CIO operations until the circular buffer can contain no more PRU's:

   (LIMIT) - (IN) < (PRUSIZE)

5. When this occurs, a WRITE is issued to dump the circular buffer (always containing an integer number of PRUs) onto the file f2.

6. Each time the circular buffer is dumped onto f2, the IN and OUT pointers are reset to (FIRST).

7. Each time a full logical record is read, n is reduced by 1.

8. COMBINE will continue to issue READ's from f1 until n goes to zero.

9. When n goes to zero, the circular buffer will be dumped onto f2 for the last time with a WRITE.

10. A write end-of-record onto f2 at level zero will be issued by COMBINE to terminate the output.

11. An END RUN request is made by COMBINE.

An ABORT will be made by COMBINE should an I/O error occur during this operation.

14.3 COMPARE

General

COMPARE is a CP program, about 5600B words long. It is not a subroutine, and is expected to be called only by a control card; it can terminate only with an ENDRUN or an ABORT.

Function

COMPARE will compare two files from their present positions until a given number of e.o.r.'s of a given level number have been read through. Comparison is abandoned if a level number difference in corresponding e.o.r.'s is found. The control card may specify that after a given number of conflicting records are found, comparison will be abandoned.

A final dayfile message is given whether the total comparison was identical ("GOOD COMPARE") or not ("BAD COMPARE").

A message is printed for every conflicting record pair and if the length of the two records is different, the difference will be printed. The control card may specify that the first n conflicting pairs of words in each conflicting record pair will be printed in octal.

Entry Information

Only what has been taken from the COMPARE control card and stored in RA+2 ff. Bits 0-17 of the word at RA+64B contain the number of parameters that were taken from the control card; there must be at least two, but any beyond six are disregarded.

RA+2 contains one file name, which we shall move to the FET at RED.

RA+3 contains the other file name, which we shall move to the FET at BLUE.

RA+4 may contain the number of e.o.r.'s of a given level through which comparison is to continue. If so, it is converted from decimal to binary and replaces the default value 1 in cell HOWMANY.

RA+5 may contain the level of the e.o.r.'s to be counted according to HOWMANY. If so, it is converted from decimal to binary and replaces the default value 0 in cell LEVEL.

RA+6 may contain the number of conflicting word pairs to be printed out for each conflicting pair of records. If so, it is converted from decimal to binary and replaces the default value 0 in cell ERRORS.

RA+7 may contain the number of conflicting record pairs after which comparison is to be abandoned. If so, it is converted from decimal to binary and replaces the default value 30000 in cell RECORDS.

Exit Information

None.

Other Programs Called

CPC, IO.

Narrative

We begin at COMPARE by fetching and using the control card parameters as described above under "Entry Information". Abort if there are fewer than 2 parameters, or if any of the parameters after the first two contains a non-digit.

At VB begins the main cycle, for reading and comparing the next record pair on the two files. First set ERC and REDC = 0 and WDCNT = -1. ERC is the count of conflicting word pairs in the record, and WDCNT is always 1 less than the number of word pairs compared so far. REDC is used in deciding on parity as follows: we try to read the "RED" file in whatever parity its FET now indicates. If this does not produce a parity check, go on to VBB; but if it does, and REDC = 0, set REDC = 1, invert the parity (mode) bit in the FET, backspace, and try again. But if REDC already = 1, neither mode worked, and we go to VG to give an error message, skip to e.o.r. on both files, and go to AA to do the e.o.r. procedure.

When we come to VBB, we have read the first file without a parity check, and the FET shows which mode. Now call subroutine READBLU to read the second file in the same mode. If no parity check, go on to VK. If parity check, go to VH to give an error message, skip to e.o.r. on both files, and go to AA to do the e.o.r. procedures.

Note that if either file is on magnetic tape, the first file named on the control card should be on a tape file. Otherwise, a case like this might occur: COMPARE (DIS,TAPE), where TAPE is a BCD tape file, and DISK is an identical disk file. But COMPARE will be able to read DISK initially in the binary mode without a parity check; and, having done so, will give an error message when it cannot read TAPE in the same mode. If the control card were

COMPARE(TAPE,DISK)

all would be well.

At VK we have begun to read the corresponding records, and have found a mode that works on both of them. Now we compare them, word by word, refilling the buffers as necessary.

If a word pair does not match, we go to VC. If ERC=0, this is the first discrepancy in the record pair, so we write out the message "CONFLICT IN RECORD n", and reduce the count in call RECORDS by 1. If it has become negative, write the message "CONFLICTING RECORD COUNT EXHAUSTED" and go to TERMIN. Otherwise, at VCA, we add 1 to ERC and set ANYERR nonzero, as a signal of bad overall comparison. Now if ERC is not above the parameter in cell ERRORS, we print out the conflicting word pair in octal. Then return to VK to go on comparing.

If reading either file to refill its buffer gives a parity check, we go to VG or VH as above.

If the record on the second (BLUE) file is exhausted first, we go to VF, or in the opposite case, to VE. Either way, we set A1 to point to the FET for the shorter record, and A2 to that for the longer one; then go to VEF. At VEF, find the number of words remaining unmatched in the long buffer. Then read to the end of the long record, adding the lengths of successive buffersful to this count. Then print a message "RECORD n IN FILE x p WORDS LONGER THAN SAME RECORD IN FILE y" where p is the count we just accumulated, in octal, and n-1 is the number of record pairs in decimal, that preceded this one during the current COMPARE. Then branch to AA to do the e.o.r. procedure.

We come to AA when we have come to e.o.r. on both records of a pair. If we have written out any conflicting word pairs, they are grouped two pairs per print line, and we now call subroutine WTLIN to write out that print line if only one word pair has been placed in it. Then check the e.o.r. level numbers of the two files. If they are not the same, go to AAB, print the message "COMPARISON ABANDONED BECAUSE OF E-O-R LEVEL DIFFERENCE AFTER RECORD n. FILE x LEVEL p FILE y LEVEL q" where n is decimal and p and q are octal. Then set ANYERR non-zero, to record a bad overall comparison, and go to TERMIN.

If the two e.o.r. levels are the same, and they are below the threshold parameter in cell LEVEL, go to AAD directly. If not, count down the parameter in HOWMANY. If it is now 0, go to TERMIN. If not, go to AAD. At AAD, call subroutine UPRCNT to add 1 to the decimal record count we keep at RECCNT for use in messages. This count ignores level numbers. Then return to VB to start on the next record pair.

We come to TERMIN when we have finished or abandoned comparison. First we call subroutine WTLIN to write out a word pair, if there is one in the buffer waiting for a second pair to fill the line. Then call CPC to write end of record on our OUTPUT file. Then call CPC to write a dayfile message "GOOD COMPARE" if ANYERR contains zero, "BAD COMPARE" otherwise. Finally, call CPC to request MONITOR to end the run.

Subroutines

SUBR

This is called at the beginning of subroutines READ, WRITE, BAXP, and REDUMES to step their exit addresses by the number in the calling sequence to SUBR, and to get A1 to point to the first parameter in the calling sequence to READ, WRITE, BAXP, or REDUMES, thus putting the first parameter in X1.

Entry Information

Only the calling sequence:

```
+    RJ    SUBR
-    EQ    n
```

where n is the number of parameters in the calling sequence for the subroutine that called SUBR. SUBR assumes that RJ SUBR is in the second word of the subroutine that calls it.

Exit Information

The address of the first parameter in the call to the subroutine that called SUBR is in A1, and the parameter is in X1. The return address in the outer subroutine has been increased by n from the call to SUBR.

Subroutines Called

None.

Registers Destroyed

A2, A6, X2, X6

READ

This is used for all reading of the files being compared. It deletes any e.o.r. status bit that may be in the FET, and sets the FET pointers to empty buffer. Then it calls CPC to read with recall. Then if the error bit for a too-long tape PRU, or the end of information bit is set, go to BIGREC or EOI, write an appropriate message, set ANYERR non-zero to indicate a bad overall comparison, and go to TERMIN. Otherwise, exit from READ with X1=0 if the parity error bit was not set, or X1=1 if it was set.

Entry Information

The calling sequence

```
RJ READ
VFD 60/a
```

where a is the address of the first word of the FET. Bit 1 of the first word of the FET determines the mode.

Exit Information

X1=0 if the read took place without a hitch; X1=1 if the only hitch was a parity check. The FET will show status read completed, e.o.r. completed, or e.o.f. completed, with the level number if e.o.r.

Subroutines Called

SUBR, CPC.

Registers Destroyed

A1, A2, A6, A7, X1, X2, X6, X7

READBLU

This is used for most reading of the second (BLUE) file. It merely sets the mode bit of the second file FET to the same as that in the first file FET, and then calls subroutine READ.

Entry Information

None.

Exit Information

As for subroutine READ.

Subroutine Called

READ.

Registers Destroyed

A1, A2, A6, A7, X1, X2, X6, X7.

WRITE

This is called for all writing of messages on file OUTPUT.

Entry Information

The calling sequence

```
RJ WRITE
VFD 30/a.30/b
```

where the message is in cells a through b-1.

Exit Information

None.

Subroutines Called

SUBR, IOWRITE

Registers Destroyed

A1, A2, A6, X1, X2, X6.

BAXP

This is called only once, to backspace the first file before trying to read the beginning of a record in a second mode.

Entry Information

The calling sequence

```
     RJ BAXP
     VFD 60/a
```

where a is the address of the first word of the FET.

Exit Information

None.

Subroutines Called

SUBR, CPC.

Registers Destroyed

A1, A2, A6, X1, X2, X6.

FREEZE

This is called at VKD and VC when comparison of words is interrupted by either buffer being exhausted, or by a discrepant word pair. It saves B2, the updated OUT pointer of the first file, in RED+3; B4, the updated OUT pointer of the second file, in BLUE+3; and X7, one less than the number of word pairs compared so far in the current record pair, in WDCNT.

Entry Information

Things to store in B2, B4, X7.

Exit Information

None.

Subroutines Called

None.

Registers Destroyed

A6, A7, X6.

REDUMES

This is called to print a message whenever a record on either file gives a parity check, except that the beginning of a record on the first file is read in both modes before an unavoidable parity check is accepted.

Entry Information

Only the calling sequence

```
RJ REDUMES
VFD 60/a
```

where a is the address of the first word of the FET.

Exit Information

None.

Subroutines Called

SUBR, BLSTUF, WRITE

Registers Destroyed

A1, A2, A6, X1, X2, X3, X6.

OCTAL

This converts a word in X,regarded as 20 octal digits, to 20 BCD characters in X6 and X7. To get 20 display code characters, one must add a word of display code zeros to each of X6 and X7; this is not done by OCTAL.

Entry Information

A word in X1.

Exit Information

The same word converted to 20 BCD octal digits in X6 and X7.

Subroutine Called

None.

Registers Destroyed

X0, X1, X2, B2.

WTLIN

This writes out the line stored in LINBUF through LINBUF+9 for printing, unless it is already empty. Then it sets the pointer at LINPOS = LINBUF, indicating an empty buffer. This buffer is used to store discrepant word pairs from a record pair, two at a time. After one has been stored, the pointer in LINPOS = LINBUF+5. After two pairs have been stored, the pointer = LINPOS+10; then WTLIN is called immediately. It is also called at the end of a record pair, to clear out a possible remaining half-line.

Entry and Exit Information

None.

Subroutines Called

IOWRITE

Registers Destroyed

A1, A6, X1, X6.

UPRCNT

This is called only at AAB, when one record pair has been finished and the next is about to be begun, to increase by 1 the record count in RECCNT, for messages. This is a 9-digit decimal number, with leading zeros replaced by blanks, followed by a blank as tenth character. Its initial value is 1.

Entry and Exit Information

None.

Subroutines Called

None.

Registers Destroyed

A1, A7, all X-registers.

BLSTUF

This is called, with the first word of an FET in X1, to extract the file name from the word,insert trailing blanks, and leave it formatted for a message in X6.

Entry Information

The FET first word in X1.

Exit Information

The file name, formatted for a message, in X6.

Subroutines Called

None.

Registers Destroyed

X1, X2, X3.

14.4 COPY

This routine copies a file past two consecutive file marks. The copy will also terminate on end-of-information. Its one subroutine is used to call IO. The procedure is as follows:

1. The input file is opened with an open alter request (OPE function code -120B).
2. The output file is opened with an open write request (OPE function code 104B).
3. The input file is copied to the output file until the end file status (30B) has been returned by CIO two consecutive reads. Both EOF's are copied to the output file.
4. A SKIPB of one record (CIO function code -640B) is issued for each file, thus backspacing over the second EOF.
5. An END request terminates processing.

14.4 COPY

This routine copies a file past two consecutive file marks. The copy will also terminate on end-of-information. Its one subroutine is used to call IO. The procedure is as follows:

1. The input file is opened with an open alter request (OPE function code -120B).
2. The output file is opened with an open write request (OPE function code 104B).
3. The input file is copied to the output file until the end file status (30B) has been returned by CIO two consecutive reads. Both EOF's are copied to the output file.
4. A SKIPB of one record (CIO function code -640B) is issued for each file, thus backspacing over the second EOF.
5. An END request terminates processing.

14.5 COPYBCD

General

COPYBCD is a CP program, about 2200B words long. It expects to be called only by a control card; it is not organized as a subroutine, and ends only with an ENDRUN or ABORT.

Function

The control card may have from 0 to 3 parameters - none, or a number, or one or two names, or one or two names followed by a number. The first name is for the input file; the second is for the output file; the number is the number of files to be copied. The default names are INPUT and OUTPUT, and the default number is 1.

The input file is assumed to be binary, containing long records of lines ending with final zero bytes. Every word whose last byte is 0 is considered to be the last word of a line. If the last word of a record does not so end, an end of line is assumed after it.

The output file is written in BCD, presumably on tape for off-line listing. Every line is written as a physical record of 148 characters. Trailing zeros in the last word of a line are replaced by blanks; then blanks are added on the right up to a total of 147 characters. The 148th character is 00B, the e.o.r. level number. If a line is longer than 140 characters, it is taken as more than one line from left to right, 140 characters per line.

Each end of file on the input is turned into two short records on the output - the first is one 1 character, for a page skip, followed by six blanks and final 00B. The second is seven blank characters followed by 17B.

Entry Information

The number of parameters is in bits 0 - 17 of RA+64B. If there are no parameters on the control card, the default values are INPUT, OUTPUT, 1. If the first parameter is a number it is the number of files to copy. Further parameters are ignored.

If the first parameter is a name, it is the name of the input file. If the second parameter is a number, it is the number of files to be copied. Further parameters are ignored.

If the second parameter is a name, it is the name of the output file.

A third parameter must be the number of files to copy.

Exit Information

None.

Other Programs Called

CPC.

Narrative

Execution begins at COPYBCD. First the parameters are fetched as described above under "Entry Information", and stored at INPUT, OUTPUT, and COUNT. If a parameter that ought to be the number of files contains a non-digit, we abort with a message.

At CBBA we begin each file by setting B5 to BBUF, the start of the output buffer, and B6 to BBUF+14, the last+1 word we can fill. These values are kept throughout the processing of a file, and are only changed for writing the e.o.f. At CBB, we set IN=OUT=FIRST in the input FET, and then call CPC to read the input file. If the read produces e.o.f. status, branch to EOF. Otherwise set B2 to the input OUT pointer, and B3 to the input IN pointer, and begin moving words from the input to the output buffer. We use B4 to point to the next empty word in the output buffer.

When we have fetched from the input buffer a word that ends in a zero byte, we go to DC, replace trailing zeros by blanks, and store the word in the output buffer. Then go to DD to write the output record.

When B4=B6, the output buffer is full, though we did not find a word ending in a zero byte, and we go to DD to write the output record.

At DD, we call subroutine WRITE to write the output record. The output buffer, which is initially full of blanks, is refilled with blanks after writing. Then set B4=B5, i.e. to the start of the output buffer, and go to DE. Here we step the input OUT pointer in B2, which was not stepped after we fetched the last word of the preceding line. If now B2=B3, the input buffer is empty and we go to EOB. Otherwise back to DA to start filling the output buffer again.

We may also come to EOB from DE in the middle of an output line. At EOB, we save B4, the output IN pointer, in its FET. Then see whether the input FET status is e.o.r. If not, go back to CBB to read more and continue. But if e.o.r., zero the e.o.r. bit so that CPC will allow us to continue reading later, and see if B4=B5, i.e. if the output buffer is empty. If so, go back to CBB to continue. But if not, call subroutine WRITE to write out the output buffer, and then go back to CBB.

On reading e.o.f. on the input file, we go to EOF. Presumably the input and output buffers are empty, and we have completely copied out the preceding file. We store a page skip control character at B5, the start of the output buffer; then set B6, the output IN pointer, to B5+1 instead of its normal value of B5+14. Then call subroutine WRITE to write the page skip as a one-word record. Then set IN=OUT=FIRST in the output FET and call CPC to write end-of-file. This, if printed from the tape, will appear as one or two odd characters at the top of the first page of the next file. Then count down on the file count in COUNT. If it is now zero, do an ENDRUN. Otherwise, return to CBBA to process the next file.

Subroutine

WRITE

This writes out the output buffer as a logical record. First the IN and OUT pointers in B6 and B5 are put into the FET. Then CPC is called to write e.o.r. Then the IN pointer is reset to B5, the start of the buffer, and the whole buffer is refilled with blanks. After exiting from WRITE, B4 must be reset to the IN pointer (=B5, i.e. FIRST).

Entry Information

The output IN and OUT pointers in B6 and B5.

Exit Information

None.

Subroutines Called

CPC

Registers Destroyed

A1, A6, X1, X6, B4.

14.6 COPYBF

14.6.1 Purpose

This routine has four entry points - COPYBF, COPYCF, COPYBR, and COPYCR. Therefore, it may be used in order to copy any number of records or files in either binary or coded mode. This routine can also be used to copy past two consecutive file marks. Both EOF's are copied to the output file then a backspace over the second EOF is issued to the output file.

14.6.2 Initial Entry

A. Depending on the entry point entered, flags are set to the following:

```
FRFLG   = 20B if this is a record copy
        = 30B if this is a file copy
FUNC    = 10B if coded mode selected
        = 12B if binary mode selected
FRSTRD  = 1 if this is the first read
        = 0 after the first read is executed.
```

B. The size of the buffers is dependent upon the field length declared on the job card. Buffer A starts at the last instruction of the copy routine and ends at field length minus last instruction of copy routine, all divided by two. Buffer B starts at the end of Buffer A plus one and ends at the specified field length. The size of the individual buffers is saved in cell MLRS in case the input or output file is L tape.

14.6.3 COPYLAB

A. Prior to opening the files, a check is made for the presence of the "L" parameter on the copy control card. If the parameter is absent, the copy proceeds to open the files. If the parameter is any other value except L, the copy is aborted and a diagnostic is entered in the dayfile. After it has been determined the L parameter is present and correct, the next record on the file name INPUT is read into buffer CARD. This record should contain the copy label card.

B. A check is made for the correct parameter selections. If any of the parameters are incorrect, the copy is aborted and a diagnostic is entered in the dayfile.

C. At this time the information on the label card is right or left justified, depending on the parameter, and stored to the corresponding area of the input FET. Control now goes to OPEIN.

14.6.4 OPEIN - CHKOUT

A. At this time the input file is opened using an open alter function code of 120B. The last four words of the input FET are moved to the last four words of the output FET and the output file is then opened using an open write function code of 104B.

B. After the files are opened, a check of the device type and format declaration is performed to determine whether or not the input and output formats are compatible. If the file formats are not compatible, a flag is set that later produces a message notifying the user of this condition. (Refer to matrix in SCOPE Reference Manual, Section 10.1.) If the input is a coded X tape and a record copy is selected, the size of the buffers is changed to a 14 word maximum.

14.6.5 ENTWRT - CHK5

A. At ENTWRT the first parameter, which is the input file name, is merged with the proper CIO code for a read then stored into the first word of the input FET. A return jump to CALL is performed where the read without recall is issued. As soon as the read function is accepted by monitor, the output FET pointers are set to the previous input FET pointers. The write function is formed by adding 3 to the previous read last code and status, and a return jump to CALL is performed where the write with recall is issued.

B. After each read and write is completed, a return jump to CHKCAP is performed. Bits 9 - 13 of the first word in the FET are checked for a device capacity exceeded status of 10B. If this status is present, a message is entered into the dayfile.

C. Upon completion of the write, a check is made on ENDFLG being one. If so, control goes to ENDWR where various flags are checked for program termination procedures or program continuation.

14.6.6 CHECK3 - EOI3

A. In this section the last read status is saved in WTFUNC. The status is checked for end of information, end of record, and end of file. Control then goes to EOI or EOR or EOF, depending upon whichever was encountered. In these sections a check is made for the type of copy (file/record) selected and appropriate flags are set. Decrementing the file or record count is also performed.

B. Prior to exiting each EOR or EOI or EOF section, a return jump to MOVRD is executed.

14.6.7 MOVRD

The MOVRD routine is entered via a return jump and is always exited through the entry. Its function is to move the values FIRST, IN, OUT, LIMIT and word 7, of the input FET to SAVRD and WORD7 respectively. These values will be used to set up the pointers for the output FET when the next write function is issued.

14.6.8 MOVHLD

The MOVHLD routine is entered via a return jump and is always exited through the entry. Its function is to set the output FET pointers to the values of buffer B, which were calculated when the copy routine was first entered. The MOVHLD routine is executed only if the first read flag (FRSTRD) is set to one.

14.6.9 ENDWR - END

This section is entered when ENDFLG is set to one. Cells WARNFLG, BKSFLG, CLSFLG, OPEFLG, and EOIFLG are checked for a setting of one. The flags set to one mean as follows:

WARNFLG — that the input and output tape formats are incompatible.

BKSFLG — that an EOF was encountered while executing a record copy, or a double EOF was encountered.

CLSFLG — that EOI status was returned to the input FET and the input and output files should be closed to process the trailer labels.

OPEFLG — that EOI status was returned to the input FET and the input file is a multi-file. Both files will be opened again if the file count is not zero.

EOIFLG — that an EOF was encountered while executing a record copy, or a double EOF was encountered. This flag is also set to one when EOI status is returned to the input FET and the input file is not a multi-file.

14.6.10 ERROR MESSAGES

ERR1 - N EQUAL INVALID CHAR
The number of records/files to copy is a zero, alpha character, or a special character on the copy control card. (Abort)

ERR2 - COPYLAB INCORRECT
The copy label control card is misspelled or the next record in the job stream is not the copy label card. (Abort)

ERR4 - ILLEGAL PARAM OPTION
An illegal parameter was found on the copy label control card. (Abort)

ERR5 - INVALID L CHAR
The character on the copy control card, signifying a label card is present, is not an "L". (Abort)

ERR6 - EOF/EOI ENCOUNTERED
An end-of-file was encountered before the record count was exhausted, or an end-of-information was encountered before the file count was exhausted. (Terminate)

14.6.11 INFORMATIVE MESSAGES

MSG7 - ****DEVICE CAPACITY EXCEEDED****
The tape physical record size is greater than the buffer on the read or write, or the physical record size is greater than what is declared as the data format on the job request control card.

MSG8 - FORMATS INCOMPATIBLE***COPY UNCERTAIN***
The tape formats selected on the input and output files can cause a possible loss of data significance. Therefore, the user is warned of this condition.

14.7 COPYL

General

COPYL is a utility program designed to update programs on a file without requiring a large number of control cards. Its operation involves the use of two input files and one output file. The names of these files are passed to COPYL as parameters in RA+2, 3, and 4. The preset values are OLDLIB, BINARY, and NEWLIB, respectively.

One file is copied from OLDLIB to NEWLIB, starting at the current position of each of the files. Each program on the file BINARY will replace the program of the same name on OLDLIB, that is, if such a program exists on OLDLIB. With this method, no extra control cards are necessary.

Method

A. The parameters are picked up and are checked for legality. If a parameter is found that is longer than seven characters or contains any special characters, the message "ILLEGAL COPYL PARAMETER" is issue, and the job is aborted.

B. An Open Alter is performed on each of the input files, and an Open Write is performed on the output file.

C. The file BINARY is rewound. Each record (program) is then read, and the name of each record is placed in a table. When EOF is reached, the file is rewound again. A zero word marks the end of the table.

D. The copying process is done as follows:

1. A record (program) is read from OLDLIB.
2. If the name of this program is of unrecignizable format or cannot be found in the table, the program is copied to NEWLIB.
3. If the name is found in the table, the following takes place:
   a. The table entry is set equal to one (= 1) in order to flag the entry as processed.
   b. The remainder of this record on OLDLIB is skipped, if necessary.
   c. The file BINARY is searched for the corresponding program, and when found, this program is copied to NEWLIB.
   d. A dayfile message is issued of the following format, where AAAAAAA is the program name:

AAAAAAA UPDATED

or

AAAAAAA UPDATED (CHIP)

4. When EOF is reached on OLDLIB, an EOF is written on NEWLIB.

E. The table is searched for any remaining names. If any are found, it means that these programs were present on file BINARY, but not on file OLDLIB. For each such name found, a warning message of the following format is issued:

COPYL DID NOT FIND ----- BBBBBBB

or

COPYL DID NOT FIND ----- BBBBBBB(CHIP)

BBBBBBB is the program name. Thus, the names of all programs on file BINARY will appear in either one of the two above messages. (CHIP), if appended to a message, indicates that the program was in CHIPPEWA format rather than SCOPE.

F. A final message "COPYL DONE" is issued, and the central processor is dropped.

Subroutines

A. GETNAME

All name checking in COPYL is done by this routine. It examines the beginning of a logical record and identifies it as one of the following:

1. Beginning of deck in SCOPE format - Word one, bits 54-59, is equal to 77B. Word two contains a name of 1-7 characters, left justified. The name is returned in X7, left justified, in bits 18-59 with trailing blanks removed. Bits 0-17 of X7 = 1.
2. Beginning of deck in CHIPPEWA format. This is identified by the absence of 77B in bits 54-59 of word 1. The name is found in word 1. X7 is returned with the name in bits 18-59, left justified with zero fill, and bits 0-17 = 0.
3. Beginning of a loader directive - If the input file (OLDLIB) contains loader directives, they must not be confused with program names. They are recognized as follows:
   a. The first word of the record does not contain 77B in bits 54-59. A loader directive would never be of this format.
   b. The name is one of the four loader directive names: OVERLAY, SEGMENT, SEGZERO or SECTION.

c. Bit 17 of the first word is set. This will be the case only if the word is part of a loader directive and not if it is the start of a program of CHIPPEWA format (which is the only other possibility at this point). This check allows COPYL to be able to replace a program of the same name as one of the loader directives.

If a loader directive is detected, X7 is returned = 0 so no entry will be made in the name table, and the loader directive record will simply be copied to the output file. The flag in bits 0-17 will be entered into the name table to make it possible for all programs on a file to appear unique, even if there are programs of the same name but opposite type.

B. CHAR

Since COMPASS allows program names to contain characters of display code 60 - 77, COPYL is intended to handle such names also. However, these characters may not be used in dayfile messages. This routine converts all such characters to blanks in the name in X7.

C. IO

This routine is used to make calls to CIO for reads and writes. On entry, B7 contains the address of the file name, and X5 contains the CIO op-code.

The file name and op-code are placed in the first word of the FET. The same FET is used for both input and output. The buffer pointers are properly set prior to entering this routine.

CIO is called to perform the request. Recall is then entered until the request is completed.

On exit, B5 will be negative for EOF status and zero for EOR status. X5 will contain bits 1-5 of the status returned by CIO.

D. REWIND

This routine issues a CIO call to rewind a file. On entry, the address of the file name is in B6.

At present, only the file BINARY is rewound by COPYL.

E. RESET

This routine initializes the buffer parameters. FIRST, IN and OUT are set equal to the beginning of the buffer. LIMIT is set equal to the field length, which is in A0. As a result, all of the field length above the COPYL program is used for the buffer.

F. CALL

This routine places a PP call word in RA+1 and waits until MTR clears RA+1. On entry, B7 contains the address of the PP call word.

Flags, Constants, Storage

OPECALL	First of three PP call words used for opening the three files.
ERMES	PP call word used for issuing the message at MESERR.
MESERR	ILLEGAL COPYL PARAMETER in display code.
NAMEMES	PP call word used for issuing the message at MESNAME.
MESNAME	"bbbbbbbbbb UPDATED" in display code. The program name is placed in the first word of this message.
DONEMES	PP call word used for issuing the message at MESDONE.
MESDONE	COPYL DONE in display code.
NFMES	PP call word used for issuing the message at MESNF.
MESNF	COPYL DID NOT FIND ..... in display code. The program name is placed in the third word of this message.
CHIP	(CHIP) in display code. This is appended to the message at MESNAME or MESNF if the appropriate program is in CHIPPEWA format.
LDS	FWA of the start of a table containing all loader directives. They are stored one per word, left justified with zero fill.
ENDLDS	LWA+1 of the loader directive table.
ABT	ABT in display code for PP call.
DONE	END in display code for PP call.
CIO	PP call word for calling CIO. Bits 0-17 contain the address of the FET (PCIO).
PCIO	Start of five-word FET used for both input and output files.
NAME	Name of program currently being replaced.
TABLE	Start of 500 (decimal) word long table used for storing the names of all programs on thefile BINARY. A zero word follows the last entry. After being processed, each entry is set = 1.
BUFFER	Start of I/O buffer.
FET1	Start of 13-word FET used for opening file OLDLIB. This area (also FET2 and FET3) is situated above the I/O buffer and is used only for the open request.

FET2	Start of 13-word FET used for opening file BINARY.
FET3	Start of 13-word FET used for opening file NEWLIB.

14.8 COPYN

1. MACROS

There are two Macros defined in the COPYN routine, CIO and JUST. The CIO macro is used to call the circular input/output buffer for most I/O calls. The first parameter is a local parameter (dummy variable) to prevent doubly defined variables, the second parameter is the address of the five word buffer area necessary for the CIO call. The JUST macro is used to left justify its one parameter.

CIO

A. Sets up (CIO ..... Buffer address)

B. Delays until address 1 equals zero

C. Writes the CIO call to cell 1

D. Delays until the I/O is finished (buffer address bit 0 equals 1), and exits.

JUST

A. B5 equals n. (the number of characters in the parameter).

B. Sets B5 equal to 6n.

C. Sets B5 equal to 60-6n, the number of positions to move the X register value.

D. Left shifts the XA register B5 places, and exits.

2. COPYN (Initial Entry)

COPYN first performs an open write on the OUTPUT file. Then the parameters on the control card are processed.

A. The strip flag - This parameter if non-zero sets B1 to 1 and thereby requests that all ID prefixes be eliminated from the logical records prior to writing them to the binary output file.

B. The second parameter if non-zero specifies the OUTPUT file name. If zero the error message GF7 (see last section) is sent to both the dayfile and OUTPUT and a disk file TEMP is used for the output file. If non-zero, this file is opened with an open write.

C. The third through k parameters are the input files. Input files not declared in this list will not be searched when P3 is zero or blank (see section TESTP3). Here, there are two limitations:

1. No input files - This sends the message, GF9, to both the dayfile and OUTPUT and uses TEMP as the first input file name.
2. Too many input files - Sends the error message, GF6, to both the dayfile and OUTPUT and uses only the first 10 input files declared.

At this time, each input file is opened with an open alter. When any error message is printed an ERROR flag is set and the job is eventually terminated by an ABT request. However, prior to terminating, COPYN attempts to process all its text cards.

Should the parameters on the COPYN control card be correct, the COPYN parameterprocessing loop merely sets the output file name in the five word buffer area OUTPUT.

3. READ1-READ

   The READ section sets the first input file name in FILEIN1 and the READ macro call reads 2000_8 words from the next record of INPUT to the BUFFERA area and begins translating these as text cards. The buffer size is determined by an EQU 1024 of BUFF. Changing this value will vary the number of text cards as well as the binary record buffer.

4. PROC1 - PROC2

   The processing of the text card buffer begins at PROC1 and each request for a new card returns to PROC1. This is the only area where IN and OUT, the BUFFERA pointers, are changed. When all cards have been processed, a jump to ENDBUF is taken. For each card image a return jump to BRKDWN is executed with A5 (address of the last word of the card image), P1 (the first parameter of this card), P2 (the second parameter on this card), P3 (the third parameter), and X0 (non-zero when an error was encountered) as output.

   Upon return from BRKDWN, START and END are set as the first and last words of this card image. These words are moved to BUFFERC+1 allowing BUFFERC to be a word of blanks for the carriage control. If the card image is contained in only one word the PROC3 loop is bypassed and two words are printed. If there is more than one word, the PROC3 loop transfers these words to the buffer and PROC5 increments the number of words by two (counting the word of blanks and the first word stored).

   Next, the five word buffer area, CARDS is set up to reflect the BUFFERC contents and a call to CIO sends this information to OUTPUT using PROC4 as the dummy variable for the CIO macro call. PROC2 updates the OUT pointer to reflect the first word address of the next card image. Before processing this card a check is made on X0 (returned from BRKDWN). If this register is non-zero, an error message is printed (see BRKDWN) and the next card is requested; these two tasks are handled via a jump to PROCES1.

5. PROCESS-PROCES1

In this section the first parameter is checked for SKIPR, SKIPF, REWIND, and WEOF; finding none of these, a standard text card is assumed and a jump to TEXT is taken. In the case of a REWIND card a further check is made for INPUT as the file name. And in this case an error message (GF5) is sent to the dayfile and OUTPUT at PROCES1. If any of the above four control cards are found, a transfer to CONTROL is taken.

6. CONTROL - CONR

At CONTROL the second parameter which is the file name, is merged with the proper CIO code (CODE+B5) into the first word of the five word REQUEST buffer. If the request is either a REWIND or an WEOF, a jump to CONV1 bypasses the third parameter check. If the third parameter is negative, either a backspace file or a back - space record is requested.

The code is set to 640B for SKIPB. If a backspace file is requested, a level number of 17B is set. Once the code is corrected the minus sign is shifted and a call to CONVERT at CONV2 is made to convert the display code decimal number to binary. (See CONVERT). If the third parameter is positive, a check for a plus sign, which will be shifted at CONV, is made before jumping to CONV2. Once the number is converted, a return jump to CON1 executes the request and then the next card is requested from PROC1.

7. TEXT - P1FOUND

In this section the first parameter, P1, and second parameter, P2, are assumed to be record identifications or decimal numbers. Should the first parameter, P1, be a zero, an error condition is noted at TEXT2. If P1 is alphabetic, control goes to TESTP3. If P1 begins numeric a return jump to CONVERT gives either a binary number or a jump to TESTP3 for an alpha-numeric P1. If P1 is numeric, the third parameter is read and examined; a zero here sends control to TEXT1 while a non-zero stores P3 as the input file name for searching. Next, P1 is set negative indicating to REC that this record is to be skipped on file P3 or on the current file. At TEXT1 the looping to P1NUM until P1-1 records are skipped begins and when the correct number of records have been skipped P1 is set to zero indicating to REC that this record is to be written on the output file. P1NUM jumps to REC to read a record, then to EOF for an end of file check. If an end of file is encountered and the file name is not INPUT, the next record is checked for a double end-of-file. In this case the message GF15 is issued; in the case of INPUT, the error message GF10 is issued. A non end-of-file or double end-of-file goes to P1NUM1 which loops to TEXT1 if more records are to be skipped or to P1FOUND if BUFFERB is to be written on the output file. P1FOUND calls WRITE to actually transmit the record.

8. P1FOUND - ALPHA2

Upon return from WRITE, a check is made on P2 being zero. If so, control goes to PROC1 for the next card. If not, if P2 is a letter, control goes to P2ALPHA; while a special character (* ** or /) sends control to SPECIAL. If P2 begins numeric, CONVERT is called to convert the number to binary, and SPEC is cleared. At P1NUM2, REC is called to read the next record. Then EOF checks for an end-of-file before WRITE writes the record. If an EOF does not occur, control goes to P1NUM3 which decrements P2 and either loops to P1NUM2 or PROC1 (when P2 equals zero). If an EOF occurred, SPEC is read. A zero here sends control to P1NUM4. P1NUM4 decrements P2; then P1NUM5 reads the next record, checks for an EOF; and if not, writes the record before going to P1NUM3. If SPEC is not zero, a one sends control to PROC2, a two sends control to P1NUM5. If the EOF check at P1NUM5 indicates an EOF, control goes to P1NUM6. Here a non-write to double end-of-file gives a jump to P1NUM7 for an error message. The write to double end-of-file rewinds the file with a jump to CONTROL. At P2ALPHA, REC is called to read the next record, then EOF checks for an end-of-file and WRITE writes the record. If an EOF did not occur, control goes to ALPHA2. If an EOF occurred, REC and EOF determine if a double EOF was encountered. If so, control goes to P1NUM7 for an error message. If not, ALPHA2 calls COMPARE to check the ID and a non find goes to P2ALPHA to repeat the loop. When the record is found, WRITE is called to write the record and control goes to PROC1.

9. TESTP3 - TEXT2

At TESTP3, if P3 is not zero, control goes to P3OK. If P3 is zero control goes to FR2 which calls REC to read the next record. If an EOF was read, REC is called again to read the next record to check for a second EOF. If two EOF's were read, control goes to FR7. If not, control always goes to FR6.

At FR6, if the record read is P1, control goes to P1FOUND; if not, the record identification is saved in flag before going to FR2.

At FR7, a rewind request is issued and if a non-EOF record has already been read, control is sent to FR3. If not, a check for a null file (2 EOF's only) is made. If this is the case, the processing of this file is terminated by a jump to P3LIMIT. Otherwise, control returns to FR6.

At FR3, REC is called to read a record. If an EOF did occur, the next record is read and if it is a second EOF, a jump to P2LIMIT terminates the search on the current file. If no EOF did occur, the record identification is compared with P1 and with FLAG. If it matches, control goes respectively to P1FOUND or BKSPACE. If not, the search is pursued at FR3.

At BKSPACE, a request for a backspace of a logical record is issued via CON1 and control goes to P3LIMIT to search the next file.

If all the files have not been searched, the next file in the list is read. If this file is INPUT on the current file, control goes to P3INC which increments the file searched register then goes to P3LIMIT. If neither of these files are found, P3SET sets the next file search and control goes to FR2. At TEXT2 an error message is set and control goes to PROCESS. At P3OK, P3 is set to the file name and the all files searched flag is set before going to FR2.

SPECIAL

This area is entered when P2 is a special character *, **, or /- SPECIAL reads the first ENDS entry and checks for a *; finding one sets SPEC to one at SPEC1. If a single * was not found a ** is checked and finding one sets SPEC to a two. SPEC16 returns control to PINUM2. If neither of these are found, SPEC2 checks for a /. If a / is found control goes to SPEC4, if not, an error flag is indicated and control goes to PROCES1. At SPEC4, REC is called to read a record. A zero length record sends control to SPEC5 to write the record before returning to PROC1. A non zero length record is written at SPEC3 and control loops to SPEC4. Finding an EOF issues a WEOF to CON1 then reads the next record. If a double end-of-file is found, the error message is set up and control goes to PROCESS1.

CON1

The function of this routine is to process the REWIND, WEOF, SKIPF, and SKIPR requests. The actual request is already in the REQUEST buffer area. CON1 saves the REQUEST in the STORE cell in case the request has to be restored as in the case of multi record or file skipping. CON2 decrements the number of requests then sets up the CIO call. CON3 delays until cell one is free, and CON4 delays until the request is complete. If the number of requests have been decremented to zero, a return is made. If not, REQUEST is restored and control goes to CON2. For each CIO call, the CIO call word is picked up from CON5. If this was a SKIPF or SKIPB call, bit 18 will be set to 1 to indicate the skip count. When the required number of requests are completed, exit is made to CON0 to clear bit 18. Exit is then made through CON1.

BRKDWN

This routine is entered via a return jump and is always exited via the entry. If an error occurs in this routine, the X0 register is set to the error value. The inputs are pointers to the current card image and the output is P1, P2, and P3.

At BRKDWN, the storage areas are initialized to zero and the A6 register is set to store the first parameter. B6 is set to the number of characters in the text card image. BREAK1 isolates the next byte and a zero terminates the card image. If this character is a separator control goes to BREAK4. If the character is not greater than 51B, it is stored at BREAK2, and the count is decremented by one. If this is the last character of this word, the next word is read before going back to BREAK1. If a zero byte was found, control was transferred to BREAK3 and a "Two Prameters Stored" flag is set. At BREAK4 a check for a blank results in a transfer to BREAK2, while a dollar sign, considered as a valid character, sends control to TAG. A valid separator such as comma, left or right parenthesis sends control to BREAK7. If none of these, control goes to BREAK9 for an error indication. At BREAK7, a count of the number of characters in the word is made. If this exceeds seven, control goes to BREAK6 for an error indication. If not, the parameter is left justified and stored. BREAK checks to see if three parameters have been stored and either goes to BRK8 or BREAK2. At BRK8 the error flag is cleared. Then control goes to BREAK8. Here the remainder is skipped as comment. When a zero byte is found, control goes to BRKD1 to exit to BRKDWN.

CONVERT

The function of this routine is to convert the display code decimal numbers to binary. It is entered via a return jump and is returned either through the entry or to TESTP3, P2ALPHA or PROCES1. The converted number is in the X0 register.

CONVERT initializes the registers; then CNVERT1 isolates the character. Finding a zero sends control out the entry. Finding a separator sends control to CNVRT2 which error exits to PROCES1. Finding an alphabetic character sends control to CNVERT3 which left justifies the alphanumeric field then checks to see if the entry was made checking P1 or P2. P1 sends control to TESTP3 while P2 sends control to P2ALPHA. If neither P1 or P2 is found, the error message is generated at CNVERT2. All numbers are converted and added to the X0 register, and the first zero terminates the number.

ERR

Entry to this routine is made via a return jump, and the exit is always through the entry. The error flag is set to non zero so that an ABT request will end the program. The message is sent to both the DAYFILE and OUTPUT. ERR sets up the error message address obtained by MSG+(B5). This message is sent to the DAYFILE before going to OUTPUT. The CARDS buffer area is initialized and the message is transferred to BUFFERC at ERR1. The CIO macro is called to transfer this message to OUTPUT using DUM7 as the dummy variable. Then, the error flag is set and the routine is exited through ERR.

EOF

The EOF routine is entered via a return jump, and is always exited through the entry. Its function is to check the status of the input file. If an end-of-file was read B3 is set to one; if not, B3 is left unchanged. Entry at P3EOF reads the FILEIN buffer area status response. If bits 3-5 equal 3, and end-of-file was read and B3 is set before the return through EOF.

COMPARE

The COMPARE routine is entered via a return jump and is always exited through the entry. Its function is to compare either P1 or P2 to the current record ID. The B4 register is set to indicate the results of the compare. B4 is one if a 2.0 or later record was found. B4 is a minus one if a 1.1 record was found and B4 is zero on no match. Entry at COMPARE masks off the first character of the buffer. If this character is not a 77, the SCOPE 1.1 format is assumed and control goes to RCCOMP. Otherwise, the SCOPE 2.0 format is assumed and the second word of the buffer is read. Then, at RCCOMP, the record identification is saved in RCNAME and compared with P1 or P2. If no match is found, return is made via COMPARE. If a match is found, B4 is set to -1 or +1 before return.

REC

The function of this routine is to read one record. If this record is being skipped, the entire record is read. If this record is to be written, the routine is exited with the first read results in the BUFFERB ubffer. In any case, a call to COMPARE is made after the first read in order to save in RCNAME the identification of the record and compare it with P1 or P2.

Entry at REC merely sets the entry register. At REC1, the file name is inserted in FILEIN and the remaining words of this buffer are initialized. Then a call is made to the CIO macro using DUM3 as the dummy variable. The status of the request is checked, and if an EOF or end of record is found the routine is exited. If not, a check is made on the skip record register. A zero here sends control back to REC1. If P1 is negative, indicating that this record is to be written, the return is taken. If not, a check is made on register B4, which has been set by the COMPARE routine. The result of this jump goes either to REC1 (not the record) or REC.

WRITE

The WRITE routine is entered via a return jump and is always exited through the entry. The function of this routine is to write the information in BUFFERB on the binary output file. WRITE will transfer one record and will change the information only if the strip record flag is set. At WRITE, the X7 register is set to the strip flag (B1), and an entry register flag is cleared.

WRITE3 begins the end-of-record check. Finding an end-of-record, it sends control to WRITE1. If no end of record is found, the entry register flag is set. At WRITE1, the end-of-record flag is set. If the strip flag is not set, control goes to WRITE2. If it is set, a check for the 2.0 format is made. If the format of this record is 2.0, the out pointer is incremented by the number of words in the ID table. If not, control goes to WRITE4. WRITE4 clears the X7 register so that all other information in this record is copied unchanged. Next, WRITE2 calls CIO to output BUFFERB. A check is made to see if the entire record has been written. If so, the return through WRITE is taken. If not, a jump to WRITE3 continues the looping until one record is written.

ENDBUF - ENDBUF8

This section checks the status of the text card buffer. If there was an end-of-record response, control goes to ENDBUF9. If there is no end-of-record response, an error message GF12, is sent to both the dayfile and OUTPUT before going to ENDBUF9. At ENDBUF9 a check is made on the ERROR flag (see ERR) which causes a call to either END or ABT. At ENDBUF8 the call is sent to the monitor to end the COPYN run.

ERROR MESSAGES

GF1 A PARAMETER IS GREATER THAN 7 CHARACTERS

The first separator or parameter terminator appears beyond eight alphanumeric characters. This message appears for any of the three parameters.

GF2 A NUMERIC EXTENDS BEYOND AN END OF FILE

The P2 parameter causes a jump beyond a double end-of-file. That is, P2 copies more records than exist; here COPYN writes all the records, one end-of-file and then rewinds the file.

GF3 AN ID(P1) IS REQUIRED ON ALL TEXT CARDS

This message occurs when a comma or separator appears as the first character causing the first parameter to be a zero.

GF4 TEXT CARD CONTAINS AN ILLEGAL SEPARATOR

Only, blank + - / * are acceptable in addition to the alphanumeric characters.

GF5 CONTROL CARD REWIND (INPUT) IS ILLEGAL

In this case COPYN could not reposition the input file correctly; therefore, the card is rejected, the message printed, and the INPUT file is left unchanged.

GF6 TOO MANY INPUT FILE NAMES ON COPYN

The limit is ten files. COPYN will give an error message and attempt to use the first 10.

GF7 NO OUTPUT FILE ON THE COPYN CONTROL CARD

The second parameter on the COPYN control card was zero, COPYN will set a disk file, TEMP, as the output file and continue to process the control card.

GF8 FIELD IS NON NUMERIC ILLEGAL TEXT CARD

The SKIPR and SKIPF requests use CONVERT to interpret their i field; this error message is given when i is not entirely numeric.

GF9 NO INPUT FILE ON THE COPYN CONTROL CARD

Parameters three through ten on the COPYN card are zero. A disk file TEMP, is set as the only file searched when P3 is zero (exception: an existing P3 will be searched first).

GF10 BINARY RECORD MISSING FROM INPUT

P3 is the INPUT file yet the next record on INPUT is not the expected binary record.

GF11 ID NAME NOT IN INPUT FILES SEARCHED

The P1 parameter was not found in either P3 or any of the input files listed on the COPYN control card.

GF12 TOO MANY TEXT CARDS IN THE INPUT RECORD

BUFF is the size of the input buffer. If there are more TEXT cards than area allocated by BUFF, and all the cards in the buffer are processed then the error message is printed.

GF13 P2 IS NOT IN THE FILE OR IS UNDEFINED

P2 was not found in the file or began * or / (like a special character) however was not *, ** or /.

GF14 A DOUBLE EOF WAS FOUND BEFORE A /

WHEN P2 is a /, if the end-of-file is encountered before a zero length record, this message is printed and all records to the EOF were written on the output file.

GF15 A PARAMETER BEGINS BEYOND AN EOF-EOF

P1 is numeric and causes a skipping to the double end-of-file before P1 is zero. The tape is positioned at the double end-of- file.

TABLES - BUFFER AREAS

NAME	LENGTH	USE
OUTPUT	5 words	Binary output file buffer area
ENDS	3 words	P2 special characters * ** /
REQUEST	13 words	Buffer area for REWIND, SKIPR, SKIPF and WEOF
CODE	5 words	CIO codes for BACKSPACE Backspace file Skip record forward Skip file forward WEOF REWIND
BUFFERB	BUFF	Binary records
BUFFERA	BUFF	Text cards
FILEIN	5 words	Binary input file
INPUT	5 words	TEXT CARD FILE
BUFFERC	64 words	Text card buffer to OUTPUT
CARDS	5 words	CIO buffer for text cards to OUTPUT

14.9 COPYSBF

This routine copies one file in binary mode. As the beginning of each coded record is encountered, a printer control character is inserted. (Its one subroutine is used to call CIO.) The procedure is as follows:

1. The input file is opened with an open alter request (OPE function code = 120B)
2. The output file is opened with an open write request (OPE function code = 104B)
3. The input file is read.
4. The data in the input buffer is moved to the output buffer as each word is shifted right one character and the leading character inserted. The leading character for the first coded record is a one (page eject), and the leading character for the remaining coded records is a blank.
5. The output file is written. Steps 3-5 are repeated until end-file status occurs. Due to the insertion of leading characters and of full zero words when necessary, the output file will be larger than the input file. As a result of this, the output buffer will be completely drained only during end-record writes.
6. The central processor is dropped by an END request.

14.9 COPYSBF

This routine copies one file in binary mode. As the beginning of each coded record is encountered, a printer control character is inserted. (Its one subroutine is used to call CIO.) The procedure is as follows:

1. The input file is opened with an open alter request (OPE function code = 120B)
2. The output file is opened with an open write request (OPE function code = 104B)
3. The input file is read.
4. The data in the input buffer is moved to the output buffer as each word is shifted right one character and the leading character inserted. The leading character for the first coded record is a one (page eject), and the leading character for the remaining coded records is a blank.
5. The output file is written. Steps 3-5 are repeated until end-file status occurs. Due to the insertion of leading characters and of full zero words when necessary, the output file will be larger than the input file. As a result of this, the output buffer will be completely drained only during end-record writes.
6. The central processor is dropped by an END request.

14.10 CPC

General

CPC is a CP program, about 240B words long. It is loaded only when another program uses, as an external, one of its entry points - usually the one called CPC.

Function

CPC transmits requests for PP action from CP programs to monitor, and in some cases returns information in X1, after completion by the PP program.

Entry Information

The word after the RJ CPC contains the request, and CPC will return control to the word immediately after the request word.

If bit 41 of the request word is 0, the call is for some kind of file action. Then A1, on entry to CPC, must contain the address of the first word of the FET. Bits 42-59 of the request word must contain either the name of the wanted PP program, or one of the following:

000001B	if only a recall on the file is wanted
000007B	for close or evict
000004B	for open
000002B	for read or write (without e.o.r.)
000003B	for other functions

Bit 40 of the request word is 1 if the CP is to go into recall until bit 0 of the first word of the FET is changed to 1, and 0 otherwise.

Bits 0-17 of the request word contain the function code for the file action, or 777777B for a file recall (i.e. the CP is to go into recall until the addressed FET shows the file is not busy). The remainder of the request word is normally zero, except that it may contain a record count in bits 18-35 for certain skip functions.

If bit 41 of the request word is 1, the requested action does not concern the file.

The contents of A1 are ignored.

Bits 42-59 of the request word must contain the name of the wanted PP program.

Bits 0-17 contain the address in CM at which the PP program is to start finding information and/or placing its response.

In every PP program callable by CPC, except MSG, things are arranged so that bit 0 of the word this address points to is initially 0, and becomes 1 on completion. See the CPC assembly listing, beginning at "MSG", for how MSG is taken care of. Bit 40 is 1 if the CP is to go into recall until bit 0 of the word bits 0-17 point to becomes 1; otherwise bit 40 is 0.

Exit Information

CPC furnishes exit information only after a request for file action. Then, on the return from CPC, X1 contains 0 unless the first word of the FET was left by a PP program with non-zero in bits 9-13. In that case, X1 will contain that word as the PP program left it. The first word of the FET is left unchanged by CPC except in the case of a branch to "OWNCODE".

Other Programs Called

Apart from calling PP programs, which is the whole purpose of CPC, the only programs outside itself that it may call are those provided by the program that calls CPC, as "OWNCODE" subroutines.

Bits 9-13 of the first word of an FET are a 5-bit number that could range from 0-37B. The normal value is 0. When CPC finds that the value is not zero, it may branch to an "OWNCODE" subroutine as follows:

a. if the value is 20B or more, indicating a system difficulty (such as no room for a new FNT entry), and there is an OWNCODE for "error" address in the FET, then CPC puts the first word of the FET in X1, zeros bits 9-13 of the first word of the FET, and branches to the OWNCODE.

b. if the value is 10B or 04B, indicating a hardware difficulty, CPC proceeds as for a.

c. if the value is 01B or 02B, indicating end of information or end of reel, and there is an OWNCODE for "eoi" address in the FET, then CPC puts the first word of the FET in X1, zeros bits 9-13 of the first word of the FET, and branches to the OWNCODE.

d. if the value is 05B, 06B, 11B, 12B, 15B, or 16B, indicating a combination of b. and c. then:

    1. If there is an OWNCODE for "error" address in the FET, CPC puts the first word of the FET in X1, zeros bits 11-13 of the first word of the FET, and branches to that OWNCODE.

    2. If there is no OWNCODE for "error" address in the FET, but there is an OWNCODE for "eoi" address in the FET, CPC puts the first word of the FET in X1, zeros bits 9-10 of the first word of the

FET, and branches to that OWNCODE. In branching to an OWNCODE, CPC assumes that the address given in the FET is the first word of the routine, and branches to the second word of the routine. Before doing so, it stores the exit word from CPC itself in the first word of the owncode routine. Thus if the OWNCODE is terminated by a branch to its own first word, it will be as if the OWNCODE were called by CPC as a subroutine immediately after the condition was seen, and the exit back to CPC were followed by the exit from CPC. However, all the registers except A1, X1, A6, X6 are restored, and the first word of the FET as it was before clearing any flags is loaded into X1, before going to the OWNCODE instead of just before the final exit from CPC.

Registers Destroyed

A1, X1, A6, X6. A7 and X7 are never referenced. All others are saved and restored.

Narrative

CPC is entered as a subroutine at CPC. CPC49, the response word, is zeroed. Then subroutine CPC03 is called to save all registers except A1, X1, A6, and X6 in cells CPC091 through CPC091+8. Then the exit address at CPC is increased by 1, and the address of the request word is put into A2. If bit 41 of the request word is 0, we go to CPC3 for some sort of file action. Otherwise, if bit 40 is 0, for no recall, go straight to CPC05, where we put the request word in X2, zero bit 41 and call subroutine CPC01 to pass it to monitor. Then we arrive at CPC0, where we call subroutine CPC04 to restore the registers that were saved on entry to CPC. Then pick up the response word in X1 (it must still be zero) and exit from CPC.

If bit 41 is 1 and bit 40 is 1, we have a non-file action with recall. We check the PP program name in the request word; if it is not "MSG", go ahead at CPC05 to set B2 to the address in the request word; it will be needed by subroutine CPC02 in handling the recall. Then put the request word, with bit 41 zeroed in X2, and call subroutine CPC01 to pass it to monitor. Then, having seen that the request is with recall, call subroutine CPC02 to wait until completion. Then exit from CPC via CPC0.

If we have a non-file action with recall, and the PP program name is "MSG", we branch to MSG before getting to CPC05. As the address in the request word points to the beginning of the message, it cannot be used to determine completion. So we put the address of the message in bits 30-47 of cell MSGRESP, and alter the request we shall pass to monitor so that it points to cell MSGRESP. PP program MSG knows that if bit 40 in the request word is 0 (no recall), the request word points to the message. But if that bit is 1 (recall) the request word points to a word in C.M. (MSGRESP in fact) in which bits 30-47 point to the message, and bit 0 must be set to 1 when the message has been picked up.

We come to CPC3 for a file action request. B2 contains the address that was in A1 on entry to CPC; this must be the address of the first word of the FET for the file. A2 contains the address of the request word. If bits 54-59 of the request word are non-zero, we go to CPC312, assume that bits 42-59 are the PP program name. Then call subroutine CPC4 to wait until the FET status is not busy, and process any previous response. Then go to CPC3107 to issue the request.

If bits 54-59 of the request word are zero, we regard bits 42-59 as 2X+Y, where Y is a bit which, if 1, calls for waiting for a non-busy FET before proceeding further. In that case, we call subroutine CPC4 to wait, and to handle any previous error or e.o.i. response. X must be a number between 0 and 3. If 0, this request was merely for a recall on the file, and we now exit from CPC via CPC0. Otherwise, X selects one of the PP program names (CIO, OPE, and CLO) in table CPC319.

Now we check the function code in the request word. If it is a simple read or write, and the FET shows that the file is already busy with the same action, there is no need to issue a new request merely to continue it, so we go straight to CPC3102. Otherwise, we construct in X2 a request to be passed to the monitor, and, at CPC3107, call subroutine CPC5 to issue it. Then arrive at CPC3102. Here, if bit 40 of the request word is 1, we call subroutine copy to wait until the FET shows completion of the new request, and to process any response from it. Then exit from CPC via CPC0.

If the request is for a simple read or write, and the file is not now busy with the same action, we go to CPC325. There we check against two possibilities.

a. that there is not enough space/information in the buffer to allow one PRU to be read/written. (If the FET contains minimal information, it appears to give a PRU length of 0. Then this test never prevents a call on CIO. Time may be wasted, but CIO will get the same test made more thoroughly.)

b. that the request is for reading and the FET status is e.o.r. or e.o.f. CPC will ignore a read request if the file status is e.o.r. or e.o.f. This is because if a user program is reading without re-call, it surely wants reading to stop on an end of record. But unless the user is forced to clear the e.o.r. bit before continuing, CPC cannot be sure the user has noticed the end of record.

If either of these conditions holds, we exit from CPC via CPC0. Otherwise, go to CPC3107 to issue the new request.

<u>Subroutines</u>

<u>CPC03</u>

This is called on entry to CPC to save all registers except A1, X4, A6, X6, A7, X7, and to set B1=1. It can also be called for the same prupose by any CP program, and IO and IORANDM do so.

Entry Information

The calling sequence is:

```
RJ CPC03
VFD 60/a
```

The registers will be saved in cells a through a+8.

Exit Information

B1 = 1.

Subroutines Called

CPC002.

Registers Destroyed

A1, X1, A6, X6.

CPC04

This is called just before exiting from CPC to reload all registers except A1, A6, A7, X1, X6, X7. It can also be called for the same purpose by any CP program, and IO and IORANDM do so.

Entry Information

B1 must contain 1. The calling sequence is:

```
RJ CPC04
VFD 60/a
```

Exit Information

All registers except A1, A6, A7, X1, X6, X7 have been loaded from cells a through a+8, on the assumption that they have previously been saved there by subroutine CPC03.

Subroutines Called

None.

Registers Destroyed

A1, X1, A6, X6.

CPC002

This subroutine is called only by subroutine CPC03, to format three short-register contents into one word and store it at the next available position in the storage space.

CPC01

This subroutine is entered with a CP request for monitor in X2. It waits until RA+1 contains zero then puts the request there, and then waits until RA+1 contains zero again.

Entry Information

The request in X2, 1 in B1.

Exit Information

None.

Subroutines Called

None.

Registers Destroyed

A1, X1, A6, X6.

CPC02

This waits until an FET shows non-busy status, or until a PP program has signalled completion. The signal is that bit 0 of the first word of the FET,or of the word to which the non-file-action request pointed, becomes 1.

Entry Information

B2 contains the address from bits 0-17 if a non-file-action request, or the address that was in A1 on entry to CPC, if a file-action request.

Exit Information

None.

Subroutines Called

CPC01, to send"RCL" requests to monitor.

Registers Destroyed

A1, X1, X2.

CPC5

This formats and issues (through subroutine CPC01) a request for file action.

Entry Information

B7 points to a word in which bits 42-59 are the name of the wanted PP program - either the request word in the calling program, or a word in table CPC319.

A2 points to the request word, and B2 to the first word of the FET. Bits 0-17 of the request word contain the code for the wanted function, with perhaps a level number. These are moved to the first word of the FET, but leaving its mode bit (bit 1) unchanged. Bits 18-35 of the request word contain zero or a record count; these become bits 18-35 of the word that subroutine CPC01 will put into RA+1. Bit 40 of the request word, the recall bit, is moved into the same position of the word for RA+1. B1 = 1.

Exit Information

None.

Subroutine Called

CPC01.

Registers Destroyed

A1, A4, A6, X0, X2, X3, X4, X6.

CPC4

This waits until bit 0 of the word B2 points to (the first word of an FET) is 1. Then it examines bits 9-13 of that word (error and e.o.i. flags). If all are zero, CPC4 exits. Otherwise, it copies the word into CPC49, the word that was zeroed when CPC was entered, and whose contents will be in X1 when CPC is exited. Then CPC4 exits unless one of two conditions is true:

1. Bits 9-13 contain an error flag, and the FET shows an OWNCODE error routine is provided.
2. Condition 1 is not satisfied, but bits 9-13 contain an e.o.i flag, and the FET shows an OWNCODE eoi routine is provided.

In either case, the relevant flag bits are cleared in the FET word, though they remain in CPC49, for the calling program to find them in X1 later, and the OWNCODE routine is entered. Note that both error and eoi flag bits might be present, but the above two rules, taken in order, describe what is done.

To enter an OWNCODE routine, we put the exit address in CPC in the first word of the OWNCODE. Then we put a branch to the second word of the OWNCODE in cell CPC. Then branch to CPC0. This means that if the OWNCODE is coded like a subroutine, and terminates by a jump to its own first word, we now restore all registers except A1, X1, A6, X6, put CPC49 (the first word of the FET as it was before we zeroed the relevant flag bits) in X1, and jump to the OWNCODE second word. When the OWNCODE terminates by jumping to its first word, we immediately, without any further restoring of registers, exit from CPC; i.e. jump to the second word after the RJ CPC that sent us into CPC on this occasion.

Entry Information

B1 = 1, B2 points to the first word of the FET.

Exit Information

1. If we do jump to an OWNCODE, but bit 9-13 of the first word of the FET were not found zero, that word has been saved in CPC49.
2. If we do jump to an OWNCODE, the first word of the FET has been moved to CPC49 and is in X1; all registers except A1, X1, A6, X6 have been restored, so that they contained when CPC was entered, and the exit word at CPC has been copied to the first word of the OWNCODE.

Subroutine Called

CPC02.

Registers Destroyed

A1, A6, X1, X2, X6, if no branch to OWNCODE. If there is a branch to OWNCODE, see above under "Exit Information".

CPC999

This sends a message to the dayfile and aborts the job. It can be called by any CP program that names CPC999 as an external, and IO, IORANDM, COMPARE, COPYBCD, and EDITLIB do so.

The calling sequence is:

```
+   RJ   CPC999
-   JP   a
```

with the message beginning in the next word and ending with the word at a-1. The word at a will be zeroed to provide a terminator.

14.11 DMPECS

DMPECS will read ECS and format it into one of several formats for printing. Identical values on an entire line are suppressed. In the event of a calling sequence error, a message is printed and the program stops.

Calling sequence: Control card containing DMPECS(X,Y,F,LFN)
Program will dump ECS from location X' to Y' where X' is the closest multiple of 10B smaller than X and Y' is the (closest multiple of 10B greater than Y)-1. F selects which print format will be used.

F = 0 or 1 4 words in octal and in display code per line.

F = 2 2 words in octal parcels and in display code per line.

F = 3 2 words in octal bytes and in display code per line.

F = 4 2 words in octal and in display code per line.

LFN specifies the dump file; if absent or zero, file OUTPUT is assumed.

The main program is written in COMPASSS with subroutines in COMPASS and FORTRAN.

Narrative

The contents of locations 2, 3, and 4 are converted from display code to octal and stored in the blank common array in locations FWA, LWA, and TYPE respectively. CM and ECS field lengths are also preserved in locations CMFL and ECFL. Location 5 is checked and if zero, file OUTPUT is used for the dump file. If non-zero, the contents of this word are used for the dump file name. Location 2 is then set to the dump file name which is later used by other subroutines. Location 64B is set to 1 which indicates to Q8NTRY the number of arguments passed on the program call card. Since this is dummied in, it is obviously a subterfuge to cut Q8NTRY out of the processing of the original arguments and ask it to do the substituted one. A RJ to Q8NTRY is made with B1 set to the FWA of the program and B2 set to the address of a list. Q8NTRY will set up an FET and initialize it properly and also take into consideration the ASA switch and line count which have been preset in the list specified in B2. A RJ to DMPE is then executed and control does not return to this code again.

Entry Information

Arguments on the program call card are stored in consecutive locations starting at 2. CM field length is in A0 and ECS field length is in X0.

Exit Information

As in the discussion above.

Subroutines Called

Q8NTRY and DMPE

Subroutines

DMPE

DMPE first checks the CM field length and if less than 1000B, prints a diagnostic and stops at STOP75. Otherwise it sets L1 and L2 to values which are greater than 30 bits in length. LINES is set to 100 and PAGE is set to 0.FWA and LWA are saved in locations FWAS and LWAS. Next, FWA is rounded down to the nearest multiple of 10B and LWA is rounded down to the nearest multiple of 10B and 7 is added to it. The adjusted LWA is truncated to ECFL-1 in the event that it is larger than the ECS field length. If FWA is greater than LWA, a diagnostic is printed and the program terminates at STOP76. If not, TYPE is checked to see if it has a value of 0,1,2,3, or 4. If not, a diagnostic is printed and the program terminates at STOP76. If TYPE is ok, then NDX is set to indicate the number of words on a line and IFRM is set equal to TYPE, or in the case where TYPE is 0, IFRM is set to 1. DMPE then commences calling READECS to read ECS and calling PRNT1 to print what has been read. This is done for each 512 words of ECS which is to be printed until all the ECS has been read and printed. The program then terminates at STOP01.

Entry Information

Blank common variables FWA, LWA, TYPE, ECFL, and CMFL are set to their appropriate values as determined by DMPECS.

Exit Information

None.

Subroutines Called

READECS and PRNT1

PRNT1

PRNT1 sets up a DO loop to print N words in the array IBUF where N has been set by the calling routine DMPE. First, the variables on a print line are checked to see if they are equal to the last word on the previous line. If so, an informative line is printed and these values are then skipped.

If not, one of the print formats is selected and the printing is done. Additional identical lines do not cause re-printing of the informative line. This continues until all N words in the array IBUF are either skipped or printed.

The technique for skipping print lines is to check all values on the current line against the last word on the previous line, one-half word at a time. Only one-half is checked at a time to correctly handle minus zero and other oddballs. L1 and L2, which contain the left and right halves respectively, are initially set to values which are greater than 30 bits in length so that the first comparison will always be unequal, thus always printing the first line. Page overflow and header information printing is taken care of by counting the number of lines and when 55 lines have been printed, overflow is initiated. This is done in 2 places, in suppression and normal printing. The initial value of LINES is 100 which causes the first print line to initiate overflow. Page numbering is also taken care of automatically.

Entry Information

The address of the first location to be printed and the number of words in the array IBUF are passed as arguments to this subroutine by DMPE.

Exit Information

None.

Subroutines Called

SHIFT.

SHIFT

Utility subroutine used to shift a word and called by PRNT1.

If not, one of the print formats is selected and the printing is done. Additional identical lines do not cause re-printing of the informative line. This continues until all N words in the array IBUF are either skipped or printed.

The technique for skipping print lines is to check all values on the current line against the last word on the previous line, one-half word at a time. Only one-half is checked at a time to correctly handle minus zero and other oddballs. L1 and L2, which contain the left and right halves respectively, are initially set to values which are greater than 30 bits in length so that the first comparison will always be unequal, thus always printing the first line. Page overflow and header information printing is taken care of by counting the number of lines and when 55 lines have been printed, overflow is initiated. This is done in 2 places, in suppression and normal printing. The initial value of LINES is 100 which causes the first print line to initiate overflow. Page numbering is also taken care of automatically.

Entry Information

The address of the first location to be printed and the number of words in the array IBUF are passed as arguments to this subroutine by DMPE.

Exit Information

None.

Subroutines Called

SHIFT.

SHIFT

Utility subroutine used to shift a word and called by PRNT1.

14.12 EDITLIB

General

EDITLIB is a CP program, requiring a field length of 50000B words. This provides enough space for a directory model, including bodies of CM-resident programs, 30000B words long. Its field length can be altered upwards, but not downwards, automatically by letting the first function card read by EDITLIB be

LENGTH(n)

where n is a digit between 4 and 9, requesting enough field length to accomodate a directory model of n times 10000B words.

EDITLIB is not a subroutine, and is only to be called by control card, which must be

EDITLIB.
or
EDITLIB(RESTORE)

The first form causes, before anything else is done, the current CMR directory to be written out at the beginning of a common file called SSSSSST. The second form causes, before anything else is done, the current CMR directory to be replaced by what is at the beginning of common file SSSSSST. Thus EDITLIB(RESTORE) restores the running system to what it was before the last preceding "EDITLIB" control card was obeyed. But if no "EDITLIB" has been done since dead start, it will abort the job. Of course the restore cannot take place if an EDITLIB has spoiled an essential program in the system. And a restore cannot back up more than one stage; so in general it does not restore the system as it was after dead start. Note also that two EDITLIB's can go on together, if they concern only separate groups of programs (one could patch in some new tape-labelling routines while the other patches in a new FORTRAN). But afterwards, no restore can get back to the situation as it was before the two EDITLIB's started work.

EDITLIB ends with an ENDRUN if it ran without error, or with only minor errors; otherwise with an ABORT.

For a complete description of the structure of a directory, see the long comment in the EDITLIB assembly listing that precedes subroutine GETD.

Modifications

EDITLIB has been modified to generate an external reference table (ERT) to be used by the CP loader. The ERT is a library record with D.. as its name; it is mass storage resident. It is generated automatically by EDITLIB and cannot be referenced through control cards. The CP loader uses the ERT to locate entry points for unsatisfied externals without actually loading programs from the library. The CP loader can operate whether or not the ERT is present. More efficient operation results when the ERT is present.

CAUTION: Once the ERT has been established, it must be perpetuated by use of a 3.1.5 (or later) version of EDITLIB. Any attempt to use an earlier version will break the relationship between the program name table and the external reference table. Results of this action are indeterminate in nature but may manifest themselves as unsatisfied externals at load time or improper loading. Therefore, care must be exercised to preserve the continuity of the ERT.

Function

EDITLIB begins by writing the present CMR directory at the beginning of common file SSSSSST, if the control card is "EDITLIB.", or by replacing it with what is already on common file SSSSSST if the control card is "EDITLIB(RESTORE)". Then it reads the next record on file INPUT, and copies it to a local file called SSSSSSV. This record must contain all and only the function cards to control the EDITLIB run.

Beginning at location START, the main cycle of EDITLIB reads a function card from file SSSSSSV, obeys it, and returns to START. When the end-of-record is read on SSSSSSV, EDITLIB terminates with an ENDRUN. Any earlier termination because of error will be with an ABORT.

Basically, EDITLIB has two alternative jobs: to construct and write out a new system file on disk or tape, without altering the running system, and to modify the running system itself. In the latter case, the common file SYSTEM, which is a copy of the system file on tape that was last dead-started, is not altered. The CMR directory is altered, and any new programs, or new versions of existing programs, are written on a common file called SSSSSSU, to which as well as to SYSTEM, directory entries can point. (There are also local files created by EDITLIB called SSSSSSS, SSSSSSV, SSSSSSW, and SSSSSSX, so these names should also be avoided by a job that includes EDITLIB.

To begin making a new system file, one gives the function card

READY(f)

where f is the name of the file, and is not SYSTEM. To begin modifying the running system, one gives the function card

READY(SYSTEM).

To complete either process, one gives the function card

COMPLETE.

If the running system is being modified, it is not affected until the "COMPLETE" card has been obeyed.

Library revision functions are as follows:

TRANSFER, a function that can be issued to move one or more of the first fifteen records (those through DSD, which are not really part of the library) onto a new system file.

ADD, which adds a PP program, CP program, or already-formatted overlay.

ADDCOS, which adds a "STITCH" program (a 1.1 CP program in assembly binary form).

ADDBCD, which adds a record of BCD cards and turns them into an overlay.

ADDTEXT, which is like ADDBCD except that the input, probably an output from program EDITSYM, is a binary record in form.

There are two function codes that cannot be given between READY and COMPLETE: MOVE, which alters the residence of a program in the running system library; LIST, which lists the program in the running system library or in another system file.
There are four functions that can be used indifferently:

REWIND, which rewinds a file.
SKIPF, which skips forward on a file.
SKIPB, which skips backwards on a file.
LENGTH, which asks for a greater field length.

The function DELETE has two slightly different meanings, depending on how it is used. If between READY and COMPLETE, it deletes a program from the system file other than the running system directory that will become the running CMR directory when the next COMPLETE is done. Otherwise, it deletes a program immediately from the running system directory in CMR.

Entry Information

Only the control card "EDITLIB." or "EDITLIB(RESTORE)". If the latter, there must be a common file SSSSSST containing a copy of the CMR directory as it stood at some time.

Exit Information

A common file SSSSSST, containing a copy of the CMR directory as it stood when the last "EDITLIB." control card began to be executed.

Other Programs Called

CPC, IO, MDI, CTS, SRB, MSG, MEM, CIO. Note that MDI is a PP program used only by EDITLIB. If one deletes it from the system (by an EDITLIB run) after dead start, the system is protected from future EDITLIB efforts. It is still possible to make up new system files, though not to use the running system as a source of programs except by using the system tape that was dead-started as an equivalent source; but it is no longer possible to alter the running system.

Messages

Each function card, after being read from file SSSSSSV and just before being obeyed, is copied to the console display, system dayfile, and job dayfile. It is prefixed with 10 blanks to indent it and distinguish it from a system control card.

Error messages go not to the dayfile but merely to the OUTPUT file. One error message that does not cause an ABORT, but merely skips the faulty function card that caused it:

ABOVE IS ILL-FORMED AND IGNORED.

This is preceded by a reproduction of the offending card. The possible faults are:

A. It contains more than 30 elements (words and/or numbers).

B. The first element is not one of the EDITLIB functions.

C. Any other element is longer than 7 characters.

D. An element begins with two or more digits, and contains a letter. (An element is allowed to begin with one digit and contain letters, because of program names like "2TS".)

E. An element ought, for the sake of the function code, to be a name, but is in fact a number.

F. An element that ought to be a residence code is not one of the allowable ones: currently "CM" and "DS".

G. On a SKIPB card, the file name is followed by something other than a number, or is not followed by anything.

H. On a SKIPF card, the file name is followed by something other than a name or number, i.e. by an asterisk or dash.

Other error messages cause an abort, unless marked with an asterisk below. They are:

BINARY CARD READ AS FUNCTION CARD.
The first record EDITLIB reads from the INPUT file must contain all and only the function cards. If a character that is not a letter, digit, blank, or +-*/()$ comma or period is read in this record, it is assumed that a binary card has found its way into the deck at that point.

CANT MOVE WHILE READY PENDING.
* CANT LIST BETWEEN READY AND COMPLETE.
Functions MOVE and LIST are not allowed between READY and COMPLETE.

CANT TRANSFER WITHOUT READING.

CANT ADD WITHOUT READYING.
Functions TRANSFER, ADD, ADDBCD, ADDCOS, and ADDTEXT are allowed only between READY and COMPLETE.

READY TWICE WITHOUT COMPLETE.
A ready has occurred, without a following COMPLETE, and now another READY has been read.

NO READY.
A complete function card has been read, but there has been no preceding READY, or no READY since the last COMPLETE.

INPUT FILE ENDED BEFORE ((ADD)) CARD SATISFIED.
EDITLIB is trying to ADD, ADDBCD, ADDCOS, or ADDTEXT one or more programs to a system file; the input file (not necessarily file INPUT itself) has ended before the program named in the function card as the last or only program to be added has been found.

END OF FILE IMPROPERLY READ ON INPUT FILE
A transfer function card calls for copying one or more records from an input file to a system file under construction. An end-of-file has been read on the input file before all these records have been found.

SKIPF FUNCTION MET END OF FILE
A SKIPF card has called for skipping forward one or more records on a certain file. Before this request could be satisfied, an end-of-file has been encountered.

INPUT REC. MISPREFIXED FOR ((TRANSFER))
A transfer function card calls for copying one or more records from an input file to a new system file. Either the function card specified a program name, and the input record had no prefix, or has a prefix containing a different name, or the control card did not specify a program name, and the input record did have a prefix.

INPUT REC FOR ((ADDBCD)) HAS IMPROPER NAME CARD
ADDBCD required that the input record begin with a Hollerith card containing the name of the record in cols. 1 ff. If the record does not begin with such a card, or if the name is unacceptable (it must be fewer than 8 characters, and contain a letter as its first character and letters or digits as the other characters) the above message is given.

TRIED TO ADD PROGR. WITH ENTRY PT. WITH DUPL. NAME
The program now being added to a system file contains an entry point with the same name as an entry point of some program already in the file.

THIS REC. NAME ALREADY IN OUTPUT FILE - XXX
XXX represents the name of a program we are now adding to the system file being constructed; a program with the same name is already in the file. Note that ADDCOS will never produce this message, as "STITCH" programs are not checked for duplication; nor will the existence of a "STITCH" program already in the file cause the message.

CANT FIND DIRECTORY RECORDS ON INPUT FILE
A list function card has called for listing the programs in some system file ("LIST(SYSTEM)" cannot give this message.) EDITLIB reads the 2 records following DSD on the file, which should constitute its directory, but cannot find them. Presumably the file is not in fact a system file.

CANT ((ADDTEXT)) FROM SYSTEM

ADDTEXT is rather a special function, which reads a record of things like macro definitions from a file which is probably an EDITSYM output, and formats them into an overlay record for a system file. But if the input file specified on the ADDTEXT card is SYSTEM, this is impossible. Any such record can only appear there already formatted as an overlay, and is quite unsuitable as the source for ADDTEXT. A simple ADD could probably get the relevant record from the running system.

* PROGRAM NOT IN SYSTEM

A move or DELETE function card has named a program that is not in the system file EDITLIB is trying to modify.

SOURCE AND DESTINATION BOTH SYSTEM

The most recent READY function card was READY(SYSTEM). Now an ADD card is naming SYSTEM as the source of a program to be added to the current system, which is an absurdity.

BAD NAME CHECK XXX

XXX represents the name of a program just read from an input file; this is not the same as the first program name given on the ADD, ADDBCD, ADDCOS or ADDTEXT function card now being obeyed. If the source is the running system library, however, XXX will be the name of the program requested on the control card, which could not be found in the directory.

DIRECTORY UNDER CONSTRUCTION GETS TOO BIG - TOO MUCH CM RESIDENCE

EDITLIB is constructing a directory that will replace the SCOPE system directory in central memory. This has become too large for the control point field length. As this field length allows a directory at least 30000B words long, it must be because too many programs are being assigned to CM residence. In order to allow a larger directory, try EDITLIB again with " LENGTH(n) " as its first function card, where n is a digit between 4 and 9, calling for a directory model space of n times 10000B words.

EDITLIB PROGRAM FAULT IN SUBRT. SQB

This means a branch to ERRA has taken place in subroutine SQB. See the assembly of that subroutine for the ways it could happen; but it should be impossible.

MOVE ROUTINE FINDS CANT READ PROG IN SYSTEM FILE

This is a branch to ERRB, which occurs at only one point. The least unlikely way for this to happen (it never has) would be for the directory to show too long a length for a program in file SYSTEM or SSSSSSU, which we are trying to read from disk into the directory in order to make it CM resident.

EDITLIB PROGR. FAULT IN SUBRT. MAKE

This indicates a branch to ERRF. An input record started off with the proper name, but was somehow badly misformed thereafter. The only way this has happened in practive is with exchanged or missing cards, after the first one, in a CP program binary deck. EDITLIB

has to follow the division of such a CP program record into "tables", so as to extract entry point names from entry point tables; any distortion of the deck will probably cause it to end in the middle of a table, and will cause this message.

((COMPLETE)) FINDS REC. MSG. IN FILE SSSSSSS
While constructing a new system file, EDITLIB has obeyed an ADD card by writing a program record on local file SSSSSSS, and saving its disk address. Now it is trying to read it back, to copy it onto the system file itself, and the read was unsuccessful. This is a branch to ERRL, which has never happened in pratice.

EDITLIB-CTS FAULT IN INITIALIZING COMMON FILES
This is a branch to ERRP.

a. the control card was EDITLIB(RESTORE), but there is no common file SSSSSST to get the old directory from, or the file begins with something that does not look right.

b. a common file called SSSSSST was already at the control point when EDITLIB began. Note that a preceding EDITLIB run in the same job would have dropped SSSSSST from the control point before terminating, so some other program must have created it, and there is a conflict.

c. before it obeys any function cards, EDITLIB brings to the control point, or creates if necessary, a common file called SSSSSSU to contain any programs that have to be written as addenda or corrigenda to file SYSTEM. But if EDITLIB finds that some other control point has a common file called SSSSSSU, there is a conflict and it aborts. If the other control point were doing EDITLIB, it would be all right for our EDITLIB to keep trying until that EDITLIB run was finished and released the SSSSSSU file. But this cannot be the case, because one EDITLIB job does not work as long as there is an EDITLIB running at another control point. File SSSSSST provides this interlock. The last thing EDITLIB does is release common file SSSSSST from its control point. Practically the first thing it does is inquire about SSSSSST, and if there is a common file SSSSSST at another control point, it simply waits until that control point has released it.

Narrative

EDITLIB begins by storing a cautious field length in cell BENDLIM, and setting a pointer to the beginning of the directory model area in DIRPTR. Then it calls subroutine MEM to make sure there is at least room for a directory model of 30000B words. (This is the standard value; it can be altered upwards by a LENGTH function card.) Then, if the first parameter on the EDITLIB card was RESTORE, go to EDITE; otherwise to EDITA; the program from both these points will reunite at EDITM.

At EDITE we ask for common file SSSSSST, then from it read an earlier directory into our directory model space, then call subroutine RTRN to replace the current system directory with what is in our directory model

space. Then go to E49T4. If there is a common file SSSSSST at another control point, we keep asking for it until we get it. But if there is already a common or local file SSSSSST at our control point, or if there is no common file SSSSSST anywhere, we abort through ERRP.

At EDITA we ask for common file SSSSSST, and if it is at another control point, keep asking til we get it. If it is not at another control point, whether we got it or not, do not wait, but rewind it. Then we know we have just rewound an SSSSSST which was one of the following:

a. common; the proper EDITLIB SSSSSST, left over from a previous EDITLIB run.

b. local, newly created for the rewind. This is good.

c. local, left over from a previous non-EDITLIB run in the same job. This is all right unless the user thinks a run after EDITLIB is going to find that local file - EDITLIB is going to make it common and release it from the control point.

d. common and already at this control point before EDITLIB began. It must be left over from a preceding run, not EDITLIB. There is no reason why we should not use it, there is trouble ahead if some program other than EDITLIB is using common files called SSSSSST.

In cases a, c, or d, we now have a common file; in case b, a local one. We check the response from subroutine DOCTS, which looked for the file. If this is 1, the case was a or c, we now have a common file, and we go to EDITB. Otherwise we call DOCTS again for file SSSSSST. If the previous case was b, this now makes the local file common, and gives a response 1. If the previous case was d, this does nothing and gives a response 11B because we already have such a common file at our control point. In either case, go to EDITB. Abort through ERRP, however, if the response is 7, which means we did not yet have a common file SSSSSST at our control point but did have a local one, and there was a common file of the same name at another control point, or at control point 0. This must have come into existence as a common file since we arrived at EDITA; it is not another EDITLIB job that is causing the conflict, because it would wait for us to be through with SSSSSST, so rather than get mixed up with some other job we abort.

At EDITB we call subroutine GETD to copy the running directory into our directory model area, and then write it out on file SSSSSST. Then go to EDITM.

We come to EDITM when we are through, one way or the other, with SSSSSST. Call subroutine DOCTS to fetch common file SYSTEM, so that if necessary we can read disk-resident program bodies from it (we will never write on it). If the response is 3, some other control point has it, and we just keep asking. If the response is 1, we have it now and proceed. Any other response causes an abort through ERRP; someone has just destroyed file SYSTEM, or this control point already has a local or common file called SYSTEM (not because of a previous EDITLIB, which would have dropped it in terminating) and this is bad.

Then do the same for file SSSSSSU, from which we may read disk-resident program bodies, and on which we will write any addenda or corrigenda to the running program library. If the response is 3, keep waiting till the other control point releases it and we get it. If the response is 1, we have it and go ahead. If the response is anything else, rewind SSSSSSU, which should create a local file, then call subroutine DOCTS to make it common. If the response is now above 5, we abort through ERRP. Some other program is creating common files called SSSSSSU, and this is bad.

Having secured common files SSSSSST, SYSTEM, and SSSSSSU, we come to KUGEL, where we rewind file SSSSSSV. Fortunately this is simpler; we really do not care what kind of file it is as long as we can use it as a scratch file. One way or another, it now exists, probably as a newly-created local file. Now at KLUGE, we read cards from file INPUT and copy them to file SSSSSSV until we get an end-of-record response from INPUT. Then write end-of-record on SSSSSSV, rewind it, and go to START. We have copied this record to SSSSSSV, assuming that it contains all and only the function cards for this EDITLIB run. If we read them directly from file INPUT, and there were also things like binary program decks in the INPUT file as source material, the function cards would have to be mixed with the source records.

The main cycle of EDITLIB begins at START. Here we read a card from file SSSSSSV. If the response is end -of-record, we go to ENDB, put an "END" request in ENDD to be transmitted in a minute; then write end of record on the OUTPUT file, call PP program CTS to release common files SYSTEM, SSSSSSU, and SSSSSST from this control point, and then issue the "END" request.

If we get a card rather than e.o.r. from file SSSSSSV, call PP program MSG to handle the card with 10 blanks prefixed to it, copying this to the console display, system dayfile, and job dayfile. Then, in the stretch of program that begins five instructions before STD and ends at STB, we break up the card into "elements", stored in BDOWNff. The rules are:

a. a card is scanned up to and including column 80, or up to but not including the first period or)

b. every * or - is treated as an element and stored as 47B or 46B.

c. blank + / (= comma and $ are indifferently counted as separators. Two or more consecutive separators count as a single separator. A sequence of one or more separators at the beginning of a card has no effect; it is as if the card began with the first non-separator.

d. a character which is not a letter or digit, and not one of the special characters listed above, causes an abort with the message "BINARY CARD READ AS FUNCTION CARD".

e. as well as being elements in their own right, * and - act as separators to delimit preceding and following words if necessary.

f. a string of letters and/or numbers between separators is an element. If it is the first element of the card, it may be as long as eight characters, as it must be the function code, and some of them are that long. Other elements must be seven or fewer characters. If an element is too long, we give the message "ABOVE IS ILL-FORMED AND IGNORED" after a copy of the card, and go back to START for the next card.

g. if an element begins with two digits, any further characters must be digits. Otherwise, do as in e.

h. an element consisting entirely of digits is a number, and is converted from decimal to binary, complemented, and stored as a negative integer less than 2**17.

i. an element containing any letters in a word, and is stored left-justified with zero fill.

Having broken the card down we come to STB, where we compare the first element to the list of valid function codes. If no match, go to MSGA to give a message and try the next card. If match, go to the start of the corresponding function routine. Unless we abort during such a routine, we eventually return from it to START. Here are the routines:

LENGTH routine, starting at LEN

The first parameter (second element of card) must be a number between 1 and 16;otherwise go to MSGA to give a message and try the next card. Multiply the number by 10000B and put it in the calling sequence for subroutine MEM. Then call MEM to get a field length at least large enough to give a directory model space of that many words. Then return to START.

REWIND routine, starting at REW

Call subroutine MNA to check the first parameter (second element of card) for being a word, and to store it in the FET that begins at CBUF. Then call subroutine PCBUF to initialize the FET at CBUF to handle the file whose name is already at CBUF. Then call subroutine MCBUF to rewind the file named by the FET at CBUF. Then return to START.

SKIPF routine, starting st SKP

Call subroutine MNA to check the first parameter (second element of card) for being a word, and to store it in the FET that begins at CBUF. If the second parameter is missing, or is asterisk or minus, exit through MSGA. If the second parameter is a number, go to SKPE, where we repeatedly call subroutine RDR to skip forward one record, and then add one to the second parameter, which is a negative number. When it turns positive (zero), go back to START. But if the response from subroutine RDR is non-zero, we have struck end-of-file, and go to SKPB to give a message and abort.

If the second parameter is a word, we arrive at SKPC and call RDR to read-and-skip a record, i.e. read as much as possible into the buffer, and in any case pass to the end of record. Now if the record has a prefix with a name that matches the second parameter, we are finished and go back to START. Or if the response from RDR is non-zero, we have struck end-of-file and go to SKPB to give a message and abort. Otherwise return to SKPC to try the next record.

If the first word in the buffer is 7700xxxx000000000000$_B$ (this is the usual format) or 77000002000000007777B (this is the format for the first word of a program record in a system file) we can safely assume we have a prefix. Otherwise go back to SKPC and try the next record. If there is a prefix, the name of the record, to be matched with the second parameter, is in bits 18-59 of the second word in the buffer.

SKIPB routine, starting at BKS

Call subroutine MNA to check the first parameter for being a word, and to move it into the first word of the FET that begins at CBUF. Then call subroutine PCBUF to initialize the FET, using a name which we pick out of CBUF. Then check that the second parameter (third element on the card) is a non-zero number, and if not exit to MSGA. Then call subroutine MCBUF to backspace the file controlled by FET CBUF, and add 1 to the negatively-stored parameter. Do this until the parameter turns positive (zero) and then return to START. There is also a check in here for beginning of file, i.e. too many backspaces for the starting position of the file; but apparently the SCOPE system no longer gives this response anyway.

READY routine, beginning at RY

There is a cell called DESTIN, which is initially 0, and into which the READY function puts the file name given as its parameter, or 1 if the name is SYSTEM. The COMPLETE function zeroes this cell; otherwise it is left untouched.

So if the READY function finds this cell already non-zero, we abort with a message. Otherwise, call subroutine MNA to check that the first parameter (second element of card) is a word. Then match this word with "SYSTEM"; if not the same, go to RYC, where we store the name in DESTIN, rewind file SSSSSSS, and go to RYD.

SSSSSSS is a local file on which we shall keep program records for the system file we are about to construct, until the COMPLETE card arrives.

SSSSSSS is not the new system file itself. We really do not care whether SSSSSSS already exists or not; it could even be common. Just as long as we can write on it, and the rewind assures that (unless an earlier run in the same job somehow created an unwritable file called SSSSSSS and left it for EDITLIB to founder over at this point).

If the first parameter is "SYSTEM", we put 1 in DESTIN and see whether the second parameter is an asterisk. If so, we go to RYD because this means that although in principle we want to modify the running system, we want to start from scratch to build a new running system, rather than starting with the existing one and merely adding or subtracting a few programs. So the asterisk means that although the destination is SYSTEM, it is much like making up a new system file.

If the second parameter is not an asterisk, we call subroutine GETD to copy the running directory into our directory model space. This will be the basis for modifications. When the COMPLETE card comes, the modified model will be copied back into CMR. Now we return to START.

At RYD, to which we come if the first parameter is not SYSTEM, or if the second parameter is asterisk, we set up a request for PP program MDI to copy the program name table from the CMR directory, not to our directory model space, but to another space beginning at SDIR. Then call subroutine MDICALL to get MDI to do it. This means that if we are constructing a new system file, and want to copy some of the running system programs into it, we can get their disk addresses from this table and so read them from file SYSTEM or file SSSSSSU. However, if we are constructing something destined to be a new library for the running system, starting from zero instead of from the present running system (the case of READY(SYSTEM,*)) we cannot use this, because EDITLIB does not allow SYSTEM to be both source and destination. However, we could read those programs from the system tape that was dead-started. In any case, we then set up an empty directory model, to which we can add things between now and the next COMPLETE function, and then go back to START.

COMPLETE routine, starting at COM

If cell DESTIN contains 0, there has not been a previous READY, so we abort with a message.

If cell DESTIN contains 1, the READY was READY(SYSTEM) or READY(SYSTEM,*). The new programs have been written on common file SSSSSSU, where they will always be available to the running directory, and we have a new model directory in our directory model space. It would almost be enough to call subroutine RTRN to copy this directory model into the CMR, replacing the running directory, however, we have been adding programs into the model program name table by simply putting them at the end. It is necessary, before calling subroutine RTRN, to call subroutine SRT to sort the entries in the model program name table so that the order is PP programs, then CP programs, then overlays, and then STITCH programs; and within each category all CM-resident programs precede all disk-resident programs. In addition, subroutine SRT must adjust the program numbers attached to entry point names in the model entry point table, since moving entries about in the model program name table alters their program numbers. Having called subroutines SRT and RTRN, we go to COMF, where we zero cell DESTIN to show we are no longer between a READY and a COMPLETE, and then return to START.

If cell DESTIN does not contain 1, it contains the name of the new system file. We must have already, presumably by TRANSFER function cards, written out the first five records of this file (PLR, STL, CMR, MTR, DSD) and left the file positioned after the last one. It remains to write out the two records of the directory, and the program records. The programs have not been written on this file as yet, but saved on a local file called SSSSSSS; because they must follow the directory on the final system file, and yet we could not have written out the directory until we had constructed all the program records and saved them somewhere before coming to the COMPLETE card. So now we go to COMB.

At COMB, we set up FET CBUF to handle file SSSSSSS, and rewind it. Then set up FET DBUF to handle the new system file. Then call subroutine SRT to sort the model program name table into the order above described, and adjust program numbers in the model entry point table accordingly. Then, at COML, we write out the model entry point table and the model program name table, to constitute the two records of the directory in the new system file. Then we scan the model program name table from beginning to end, and for each entry, extract the disk address and put it in CBUF+6, to control the next read on file SSSSSSS; call subroutine READ to start reading the record;then call subroutine COPY to copy the whole record onto the file named in FET DBUF, which is of course the new system file. On coming to the end of the model program name table, we go to COMK, write end of file and rewind the new system file; then go to COMF to zero cell DESTIN, showing we are no longer between READY and COMPLETE, and return to START.

MOVE routine, starting at MOV.

This function cannot be done between READY and COMPLETE, so we begin by testing cell DESTIN, and if it contains non-zero, we abort with a message. Otherwise, at MOVZ, we call subroutine MNA, to check the first parameter (second element on the card) for being a word, and to move it to cell DELA. Then we pick up the second parameter, which should be the new residence symbolic code, and call subroutine RES, which returns the equivalent numerical residence code in X6, which we save at DELA+1. RES will have handled an impossible symbolic residence code by giving an error message and going back to START. But if the parameter was simply absent, RES will return a negative X6. For some functions, this would be acceptable, but not for MOVE, so we branch on X6 negative to MSGA to give an error message and go back to START.

Otherwise, we call subroutine GETD to copy the running directory from CMR to our directory model space. Then callsubroutine LOC, which scans the model program name table for a match to the program name in DELA (our first parameter). If no match, LOC gives an error message and returns to START. If a match is found, we return from LOC with the address of the entry in the model program name table in B2 and cell HOLD, and the program number for the program in B4 and cell HOLD+1. From the P.N.T. entry we extract the old residence code and compare it with the new one. If they are the same, nothing needs to be done and we return to START.

Otherwise, if the new residence code is for CM, the old one must have been for disk, and we go to MOVA to read the program from disk into our directory model. First we choose, according to a bit in the program name table entry, SYSTEM or SSSSSSU as the name of the file containing the record, and put it into FET CBUF. Now the record we are about to read begins with a 3-word prefix, and it is only what follows this prefix that we must add to the end of the directory model. So we save the last three words of the directory model in cells HOLD+5, 6, and 7, and then set out to read the record into memory beginning at the third-last word in the directory model. Knowing the length of the record, which is given (excluding the three words of prefix) in the program name table

entry, we can treat the area that will receive the new record as a work-space, whose addresses we put into CBUF+5. Then we extract the disk address from the program name table entry, and put it in CBUF+6 to control the next read. Then call IOREAD (in program IO) to read the program into the "workspace". Then restore the last three words of the directory model as it was, on top of the three words of prefix we just read at the beginning of the new record. Then increase the pointer to the end-1 of the directory model, at cell BEND, put the CM-address of the newly read program into its program name table entry, and go to MOVK.

If the new residence code is not for CM, then presumably the old residence code is for CM, and if so we call subroutine SQB to squeeze its body out of the directory model, by moving back all latter bodies, and to zero its CM-address in the program name table. Then go to MOVE. If the old residence code is also not CM, we go straight to MOVL without calling SQB. This cannot happen at present, but if a third type of residence is introduced to the SCOPE system, this particular branch will become meaningfull , though of course other changes in the MOVE routine will be needed. At MOVL, if the new residence code is DS (which at present it must be, but see the remark in the preceding sentence) we call subroutine SRB to complete the disk address in the model program name table entry. We already had a 24-bit disk address in bits 0-23 of the second word of the entry, and this suffices to find the record when the file name is known. But subroutine SRB calls PP program SRB to complete this with 21 bits more in bits 24-44 of the same word producing a 45-bit disk address which is more convenient for "PP Resident" to use when loading a disk - resident PP program. Then go to MOVL.

At MOVL we call subroutine SRT to re-sort the model program name table entries, because we have changed the residence of an entry, and entries must be sorted within each category (PP, CP, overlay, STITCH) according to residence. SRT also adjusts program numbers in the model entry point table accordingly. Then, at DELK, we call subroutine RTRN to copy the model directory into CMR, replacing the running system directory. Then return to START.

LIST routine, starting at LIS

The LIST function cannot be done between READY and COMPLETE, so the routine begins by checking that cell DESTIN contains zero, and giving a message if not. Correction: first of all, it calls subroutine MNA to ensure that the first parameter (second element of the function card) is a name, and to move it to cell DELA. Then it checks DESTIN. Now if the first parameter is SYSTEM, we need only call subroutine GETD to copy the running directory, from which we can prepare a list of programs, into the directory model space. Then go to LISC. If the first parameter is not SYSTEM, it names some other file, on which the record following DSD is presumed to be the directory. (Skipping to DSD allows the dead-start records to be variable in number as long as DSD is the last.) The record following DSD is read as the directory from which we can prepare the list by calling subroutine GROW to read the next two records and put them in the model directory space as an entry point table and program name table. Then go to LISC.

At LISC, we write on the OUTPUT file "LIST OF PROGRAMS IN FILE XXX". Then start scanning the model program name table. For each entry we write on one line of the OUTPUT file the program name, its body length (i.e. RECORD LENGTH-3) in octal its symbolic residence code, and its type, taken from table TYPE (note that type 3, "CODED INF", no longer exists in the system. This was the sort of thing ADDBCD and ADDTEXT would add to a system file, but it is now formatted as an overlay). If the program is a CP program, we look through the model entry point table for all entries with its program number, and copy out the entry point names on succeeding lines of the OUTPUT file.

When we have finished scanning the model program name table, we write a page skip on the OUTPUT file and return to START.

TRANSFER routine, starting at TRA

A TRANSFER function can only be done between READY and COMPLETE, so we first check that cell DESTIN does not contain zero, and if it does, abort with a message. Furthermore, a TRANSFER cannot be used to alter one of the first fifteen records (those non-library records through DSD) of the running system file, so we also abort if DESTIN contains 1, indicating that the last READY was READY(SYSTEM).

Otherwise, we move the file name in DESTIN to the first word of FET DBUF, which we shall use in writing onto the new system file. Then we call subroutine MNA to ensure that the first parameter (second element on the function card) is a name, and to move it to CBUF. Then we set up FET CBUF to read the file so named.

Now, just after TRAH, we call subroutine READ to begin reading the next record from the file named in FET CBUF. If the response is non-zero, we have hit an end of file, and go to TRAE to give an error message and abort. Otherwise, see whether the second parameter is absent (which will count as number zero), a number, or a name. If a name, go to TRAB. Otherwise, it was a number (zero or missing will be treated as = 1). We must copy that number of records without alternation. But each record must not have a prefix (probably we are copying off an existing system file, in which the first five records have already had their prefixes stripped off. These records originally are assembled as programs, and the binary records have prefixes; but on a system file, for necessity or convenience, the prefixes are absent.) If the record we just began reading has a prefix; that is if its first word is 770000xx000000000000B or 77000002000000007777B; we abort with a message. Otherwise, call subroutine COPY to copy the whole record onto the new system file. Then add 1 to the second parameter, which is in negative form, and return to START if it has become positive. Otherwise, return to TRAH to repeat the cycle of reading, testing and copying.

We come to TRAB when we have read the beginning of a record from the input file, and the second parameter is a name. Now we check that the input record has a prefix, and that its name is the same as the second parameter. If not, abort with a message. If it matches, we are going to copy out the record without its prefix. If the third parameter is a number greater than 1, say n, we are going to add on to the end of this record, so as to form a single output record, the next n-1 records of the input file as well, without stripping or even looking at their prefixes. This facility is provided because the first two records of

a system tape have each to be assembled partly as a PP program and partly as a CP program. The assembly program makes a separate record for each part. The two parts of each of the first two records have to be combined into a single record for convenience; the prefix has to be stripped off the first part so as to make it dead-startable, but the prefix of the second part can remain embedded in the record without causing trouble.

Now we check the third parameter. If it is absent, it will count as 0 and hence 1, and if a number it will count as itself, or as 1 if it is 0. But if it is anything but a number, we abort with a message.

Now we strip the prefix of the input record by advancing the OUT pointer of FET CBUF by the length of the prefix. Then, at TRAL, change the file name in FET CBUF to the new system file name, which we had stored in DBUF, though we shall not now use that FET. If the file status in FET CBUF is not e.o.r., go to TRALA and call subroutine MCBUF to write out. Then restore the input file name in CBUF, and call subroutine MCBUF to read more. Then return to TRAL to continue the loop. When we find an input file e.o.r. status, we add 1 to the third parameter, which is 0 or a negative count. If the result is positive, go to TRAK, where we call MCBUF to write end of record and then go to START. If the result is not positive, we go back to TRALA to continue the write-read loop; note that here we read e.o.r., but we ignore this in writing out, so as to combine more than one input record into a single output one.

ADDBCD routine, starting at ADDBCD

The ADDBCD function has practically been replaced by ADDTEXT. ADDBCD itself is an alternative to a way of using the ADD function:

    ADDBCD(MACROS,INPUT,DS)

is equivalent to

    ADD(-MACROS,INPUT,DS)

And EDITLIB was initially programmed to accept the second format (which it still accepts); ADDBCD was added to the repertory as an afterthought. So the ADDBCD routine merely moves the second and following elements from the function card breakdown one position down the line, and inserts a minus as the second element; i.e. the first parameter. Then it branches to ADD where the ADD routine begins.

ADDTEXT routine, starting at AT

ADDBCD, or the version of ADD that is its original, was programmed on the assumption that things like macro definitions for an assembly program would be a special type of record in the library (now they are not a special type, but are formatted as overlays) whose original form would be a record of BCD cards. The first card would contain the record name in column 1 ff., and the definitions would be on following cards. There would not be extraneous things like serial numbers at the ends of the cards, so normal trailing-blank suppression would compress them reasonably well.

However, it turned out that macro definitions etc. would in practice be supplied from a binary file produced as an output by program EDITSYM, in which each card image would be 80 characters, followed by a serial number in columns 80 ff., or by a row of asterisks; in the former case the length of the card-image line would be 9 words, and in the latter case 10. So to get rid of what followed the first 80 characters, and was of no use in the overlay, and to allow trailing-blanks suppression to get to the size of the overlay down, the ADDTEXT function was provided. It read in the record, ignores everything after column 72 of each line, and writes the lines, with trailing-blanks suppression, into a single record on a local file called SSSSSSW. Then it alters the parameter naming the input file from the real name to SSSSSSW, and branches to routine ADDBCD, which can then function as if the macro definitions had been supplied as a record of simply punched cards.

At AT we begin by seeing whether the second parameter, the name of the source file, is SYSTEM. If so, abort with a message, because there could be no such record in a running system; it would have been already formatted as an overlay, and in that form it could easily be added to the new system file by an ADD function. Then we call subroutine PCBUF to ready FET CBUF for reading the file named by the second parameter. Then ready FET DBUF for writing the file named SSSSSSX, then call subroutine MCBUF to rewind SSSSSSX. Presumably SSSSSSX is then brought into being by the system as a local file; at any rate it does not matter to EDITLIB if it is local or common, provided it is on disk and can be written. Now we call subroutine READ to begin reading the next record from the source file; if the response is non-zero, we have hit an end-of-file and abort with a message. Otherwise, call subroutine COPY to copy the whole record onto file SSSSSSX.

Then we call subroutine MCBUF to rewind file SSSSSSX, and then subroutine PCBUF to prepare FET CBUF to read it in BCD mode. This is the whole point of the file SSSSSSX. The source file named on the function card may have been tape; the record we have to read is theoretically binary, and if it is on tape this is an obstacle to using IOREAD in a moment, as we want IOREAD to behave as for a BCD file. So we have copied the input record to SSSSSSX, which we know to be a disk file, so that we can call it BCD when we set IOREAD to reading it.

Now we are at ATA. We copy the six-word FET model at TFET into FET DBUF, for writing a file called SSSSSSW in BCD. Then call subroutine MCBUF to rewind this file. Presumably, as with SSSSSSX, the file is created by the system at this point, but it does not matter to EDITLIB, whether it is local or common, provided it can be written. The sixth word of this FET defines a workspace for subroutine IOREAD from TLINE to TLINE+7 inclusive, or 8 words long. We now add 1 to this word and put it in the sixth word of FET CBUF, so that it defines a workspace from TLINE to TLINE+8 inclusive, or nine words long. Starting at ATB, we repeatedly go through this cycle:

a. call IOREAD to read a line from file SSSSSSX. This means a whole line (which in fact, as written by program EDITSYM, is 9 or 10 words long) is passed through, but not more then the first 9 words are put into the workspace. If the line is shorter than 9 words, the workspace is filled out with blanks.

b. blank out characters 73 to 80 in the workspace, in case they contained some sort of serial number, and would interfere with trailing blank suppression.

c. call IOWRITE to write out, as a line within the record on file SSSSSSW, the eight-word workspace beginning at TLINE. Presumably these eight words now include no serial numbers at the right end, so trailing blank suppression is effective. Then return to a.

When subroutine IOREAD returns a non-zero response, we go to ERRC to abort with a message if this indicates end-of-file (impossible) and otherwise, if end-of-record, we go to ATC. The last line in the record has already been put in the output buffer by subroutine IOWRITE; it remains only to call subroutine MCBUF to write end-of-record and rewind file SSSSSSW.

Now we have a record at the beginning of file SSSSSSW which is equivalent to the record the ADDTEXT function card told us to read, but has had serial numbers removed from the lines, and trailing blanks suppressed. So we change the second parameter, in BDOWN+2, to "SSSSSSW", and then branch to the beginning of the ADDBCD routine.

ADDCOS routine, starting at ADDCOS

Function ADDCOS adds one or more CHIPPEWA 1.1 binary CP programs to the library, as "STITCH" programs. As a signal that this is what is going to happen, the ADDCOS routine begins by setting cell COSFLAG to 1, and then branches to the ADD routine at the point where ADD has just performed its first act, that of setting COSFLAG to 0.

ADD routine, starting at ADD

The ADD routine begins by setting cell COSFLAG to 0, to distinguish it from the ADDCOS routine. After this point, the two routines join, and rely on COSFLAG for choice of action where necessary.

Now zero cells DELA+4, ONEDONE, and UPTO. These may be set non-zero later. DELA+4 will be set non-zero if the first parameter is a minus, indicating that the source file should be read in BCD mode. If the first parameter was minus, it means that the function card was

    ADD (-rec,file,res,ed)

or was

    ADDBCD (rec,file,res,ed)

which has been translated into the first format by the ADDBCD routine; in either case, file "file" must be either indifferent as to mode, like a disk file, or BCD mode, if on tape. Or the function card may have been

    ADDTEXT (rec,file,res,ed)

which the ADDTEXT and ADDBCD routines have turned into, in effect,

    ADD (-rec, SSSSSSW, res,ed)

and we know file SSSSSSW is on disk and therefore can be read in BCD.

Cell ONEDONE, if zero, indicates that we have still to match the name of the first record we take from the source file with the first program name given on the ADD function card. Once this is no longer so, either because the first or only program has been successfully ADDed, or because the first parameter is * (indicating that we start using the source file from its present position, without checking the name of the first record against the function card) then ONEDONE becomes nonzero. ONEDONE is also used, when adding more than one program from SYSTEM as source by a single ADD card, to contain the pointer to the last-used entry in the model program name table.

When UPTO contains zero, it means that the next time we finish ADDing a program, during the current function card, we will have finished processing the function card. Initially in the ADD routine, if we find that more than one program is called for, by one of the formats

```
ADD (x-y,f,r,e)
ADD (x-*,f,r,e)
ADD (*,f,r,e)
```

we put the name of the terminating program in UPTO, or put -1 upto if we are to go to the end of the current file on the source file (asterisk on the ADD card). Then whenever we ADD a program, we check its name with the terminator in UPTO, and if they match, zero UPTO.

In the case of

```
ADD (x,f,r,e)
```

UPTO remains zero throughout, as only one program is being handled.

Now, if the first parameter is an asterisk, we must have something like ADD (*,f,r,e) and we go to ADDAY, where we set UPTO to -1, to show that processing will end with the end of file on the source file (unless the source is named as SYSTEM, in which case it will end with the end of the program name table). Then we set ONEDONE to contain the address SDIR-2. This has two meanings. As ONEDONE is not zero, it means we will not check the name of the first program we read from the source file against a function card parameter. In addition, in case the source is SYSTEM, this means that by adding 2 to the address in ONEDONE, we can find the model program name table entry for the next program to be taken from the running system. A function card ADD (*,SYSTEM,r,e) would mean "take everything from the running system and add it to the new system file". So we have already set UPTO to -1, showing we are not to stop adding programs till we come to the end of the model program name table; and we have set ONEDONE to SDIR-2, indicating that the model p.n.t. entry for the next program to be taken is at SDIR. Now the last-executed READY function (provided it was not READY(SYSTEM), in which case the fact that this ADD card names SYSTEM as the source will cause us to abort with a message a little later at ADDB) has read the program name table from the CMR directory into our field length beginning at SDIR, so this setting of ONEDONE is reasonable. Then go to ADDAW.

If the first parameter is not an asterisk, we test whether it is a minus sign. If it is, we want to do the sort of thing that has been described above for the ADDBCD and ADDTEXT function routines. We set DELA+4 non-zero, showing that the source file is to be read in BCD mode. Then call subroutine MNA to ensure that the second parameter is a name, and to store it at DELA as the first and only record name to be handled. Then call subroutine MNA again to ensure that the second parameter is a name, and to store it at DELA+2 as the name of the source file. Now the fourth parameter will be the first of the residence code and/or edition number parameter, and we go to ADDAD with its address in A1 and itself in X1.

If the first parameter is neither an asterisk or a minus, we go to ADDAV. The first parameter must be the name of the first or only record to be processed, so we call subroutine MNA to check that it is a name, and to store it in DELA. Then if the second parameter is not a minus sign (type ADD (x-y,f,r,e)) it must be the name of the source file (type ADD (x,f,r,e)). So we go to ADDAW and call subroutine MNA to check that the second parameter is a name, and to save it in DELA+4. Now the third parameter must be the first of the residence code and/or edition number parameters, and we go to ADDAD with its address in A1 and itself in X1.

If the first parameter was a program name, as checked in the preceding paragraph, and the second parameter is a minus sign, we go to ADDA. Now the third parameter must be either an asterisk or the name of the last program to be ADDed (ADD (x-*,f,r,e) or ADD (x-y,f,r,e)). If it is an asterisk, we set UPTO to -1, as explained above, representing an end of file or end of model program name table; then go to ADDAC. If the third parameter is not an asterisk, we go to ADDAB, where we call subroutine MNA to check that it is a name, and to store it in UPTO as the name of the last program to be ADDed. Then go to ADDAC. At ADDAC we call subroutine MNA to check that the fourth parameter is a name, and to move it to DELA+2 as the name of the source file. Now the fifth parameter must be the first of the residence code and/or edition number parameter, so we branch to ADDAD with its address in A1 and itself in X1.

When we get to ADDAD, we have the address of the first of the residence code and/or edition number parameters in A1, and the parameter in X1. Also, we have set the following:

a. DELA+2 contains the name of the source file.

b. DELA+4 is zero if the source file is to be read in binary, or non-zero if BCD.

c. DELA contains the name of the first or only program to be ADDed, unless the function card called for adding everything between the present position and the next e.o.f on the source file. In that case, ONEDONE is non-zero, and UPTO is -1.

d. ONEDONE contains zero, showing that it is necessary to check the name of the next program taken from the source file against the name in DELA (it will be set non-zero when the check has been successfully made) except in the case noted as exception in c. above.

e. UPTO will contain zero, showing that on the next occasion that we completing ADDing a program, the function card will be satisfied; unless the function card called for adding from x to y inclusive, or for going to the next e.o.f. on the source file. In the former case, UPTO will contain the name of the last program to be ADDed, and will be zeroed when we begin to process that record. In the latter case, UPTO will contain -1.

So at ADDAD it remains only to set up the residence and edition parameters. A1 points to the first of them, if any, and from here on we work in terms of "next" parameter rather then "nth" parameter. First we set DELA+3 and REDAC negative, to indicate that the residence code and edition number have not, so far, been found. Then begin the loop for finding them at ADDAC.

At ADDADC, we check whether the next parámeter exists, and if not, go to ADDADD. If the next parameter exists and is a number, go to ADDADB, save it as a positive integer in REDAC for the edition umber, and return to ADDADC for the next parameter. But if the next parameter exists and is not a number, it should be a symbolic residence code. We call subroutine RES to look it up in table RESA, and return with its numerical equivalent in X6, which we save in DELA+4 as the numerical residence code. Then return to ADDADC for the next parameter.

When we get to ADDADD, all the parameters have been set up. If DELA+3 contains a negative number still, it means there was no explicit residence code, and we should assign anything to disk residence unless it is an already-formatted record from a system file, in which case keep its existing residence. If REDAC contains a negative number still, it means there was no explicit edition number, and we should assign number 0 to anything unless it is an already-formatted record from a system file, in which case keep its existing edition number.

Now at ADDADD, we check whether cell DESTIN contains 0. If so, we are trying to ADD otherwise than between READY and COMPLETE, and we abort with a message (after all that work on the parameters). Otherwise, go on to ADDB. If cell DESTIN contains 1, the last READY was READY(SYSTEM). Now if the source file named in cell DELA+2 is also SYSTEM, we abort with a message. Otherwise, at ADDBA, call subroutine MAKE to set up the next program; then call subroutine PCBUF to set up FET CBUT to write file SSSSSSU, and then go to ADDBAC. Remember that for READY(SYSTEM), all addenda and corrigenda to the library go to the common file SSSSSSU, which can be addressed as well as file SYSTEM by the CMR directory, and which we brought to our control point at the beginning of EDITLIB.

If cell DESTIN does not contain 0 or 1, it must contain the name of the system file we are constructing. We go to ADDC, where we call subroutine MAKE to set up the next program; then call subroutine PCBUF to set up FET CBUT to write file SSSSSSS, and then go to ADDBAC. When we are making up a new system file, we use a local file called SSSSSSS, to which we maintain an index of record disk addresses in the model program name table we are building, as temporary storage for the programs before finally writing them on the new system file when the COMPLETE function comes.

When we get to ADDBAC, subroutine MAKE has formatted a program into system file shape, and FET CBUF is ready to write it on the appropriate file. Now we store the address DELA+1 in the seventh word of that FET, so that when it is written, the disk address will be stored in DELA+1, from which we can insert it in the model program name table entry.

Now we must say a word about subroutine MAKE. Using all the parameters, it reads a record from the source file and formats it into a record ready for a system file, beginning at location PROG. Such a system file record always begins 77000002000000007777B, and the second and third words are almost like the program name table entry for the program. Then the body of the program starts at PROG+3, and the length of this body is found in bits 0-17 of the word at PROG+1. However, our buffer starting at PROG is only 4004B words long; i.e. it will be overflowed if the body of the formatted program is more than 4000B words long. If the body is not longer than 4000B words, the whole program will be laid out beginning at PROG, and cell SXCT will contain zero.

But if the body is longer than 4000B words, the first three words of the program (the prefix) will still be in PROG through PROG+2. But the body of the program will be divided up into segments 4000B words long, which have been written on local file SSSSSSX, and of which a count has been kept in cell SXCT, plus a final fragment between 0 and 3777B words long, which is still in the buffer beginning at PROG+3. We can figure the length of this final fragment, because it will be the low order 11 bits of PROG+1, i.e. the body length of the program, modulo 4000B for the 4000B-word segments that have been written on file SSSSSSX.

So just after ADDBAC, we see if cell SXCT contains 0, and if so, go to ADDBACH. Put a word in the 6th position of FET CBUF to define the whole program as the workspace to be written out, and then call subroutine IOWRITE (in program IO) to write it. Then go to ADDBACL to write end of record and proceed.

If SXCT does not contain zero, we must first write out the last of the program on file SSSSSSX. Subroutine MAKE has been using FET DBUF to write the file. We get the length of the last segment, from bits 0-10 of PROG+1, and set up the IN and OUT pointers in the FET accordingly. Then call subroutine MCBUF twice to write end of record and rewind the file. Then set IN=OUT=FIRST in the FET and call MCBUF once more to read the first segment of the body of the program. Now, beginning at ADDBACK, we are going to use subroutine IOWRITE to copy out onto file SSSSSSS or SSSSSSU, which FET CBUF is prepared to write. First we set up the word from ADDBARB at CBUF+5, to define PROG through PROG+4002B as the workspace which IOWRITE is to copy out. From PROG to PROG+2 is the prefix of the program, which has been sitting unchanged, and from PROG+3 to PROG+4002B is the first segment of the body of the program, read back in from file SSSSSUX. (As 4000B is a multiple of any PRU length, we can be sure that the last word read went to PROG+4002B. We can also be sure that this first segment was not shorter than 4000B words, because only the last segment is, and if there were only one segment, SSSSSSX would not have been used at all.)

Now call subroutine IOWRITE to move the stuff from PROG ff. to buffer BUF, which FET CBUF is set to use, and to write out some of it. At any rate we know that everything has been moved out of the PROG workspace. Now, if there is an end of record status for file SSSSSSX (this cannot happen the first time), go to ADDBACL. Otherwise, set IN=OUT=FIRST in the FET for reading SSSSSSX (we can safely do this because we must have read 4000B words into PROG+3 ff., and we have used all of them now). Then call subroutine MCBUF to read SSSSSSX again; it may read a non-last segment of 4000B words, or the last segment of 0 to 3777B words. What ever it did read, we combine the IN and OUT pointers that define it, into a word suitable for putting in CBUF to control the next use of IOWRITE, and go back to ADDBACK. There we shall again call IOWRITE to empty the SSSSSSX buffer, and continue filling and writing the SSSSSSS or SSSSSSU buffer. When we find e.o.r. status on SSSSSSX, the buffer is already empty, and we go to ADDBACL to write end of record on the output record.

To ADDBACL, whether the program was long or short, we come to write end of record on the formatted output record. When the first PRU of this record was physically written, its disk address was stored in DELA+1. So we insert this disk address in the prefix at PROG+2. We shall need it either as part of the directory model that we shall return to CMR, or as part of what amounts to a table of disk addresses for finding records on file SSSSSSS, when it is time to copy them to the new system file.

Now if DESTIN does not contain 1, the question of CM-residence does not bother us very much, and we go to ADDE. But if it does contain 1, we must now check whether the residence code for the new program is CM. If not, we need only call subroutine SRB, which we do at ADDEF, and then go to ADDE. Subroutine SRB calls PP program SRB to convert a 24-bit disk address and a file name into a 45-bit disk address for the same record, which will be more convenient for PP resident to use when loading a PP program from disk. We do not have to call SRB before going to ADDE if we are making a new system file; but if we are modifying the running system, and the residence code for the new program is not CM, we assume it is disk resident, go to ADDEF, and call subroutine SRB to complete the disk address in the prefix itself, at PROG+2.

But if DESTIN contains 1 and the residence code of the new program is CM, we have to insert the body of the new program at the end of the directory model. First check whether the body, plus the two words of the new entry point, would fit within our field length in addition to the existing directory model. If not, go to ERRQ to abort with a message. Otherwise, the present end+1 address of the directory model is obvious the starting address of where we are going to put the new body, so we fill that address into the CM-address field of PROG+2. Then increase the pointer to the end of the directory, in cell BEND, by the length of the body. Now check cell SXCT. If it contains 0, the whole body of the program is still sitting in the PROG buffer, beginning at PROG+3. So at ADDBE, we simply copy it onto the end of the directory model, and then go to ADDE. But if SXCT does not contain 0, the record except its prefix has been entirely written out previously as the

first record of file SSSSSSX. So we set BUF+5 so as to provide a "work-space" for this record to be read and moved into, which coincides with the space that the new program body must occupy at the end of the enlarged directory model. Then call subroutine IOREAD to read it, and go to ADDE.

We come to ADDE when we have written out the new program on file SSSSSSS or SSSSSSU, and dealt as necessary with the possibilities of putting its body at the end of the directory model, and completing its 45-bit disk address. Now we set bit 51 of PROG+3, which is destined to be the second word of the program name table entry, to 1; if we are modifying the current running system, this will tell any program that consults the directory that the program is to be found on file SSSSSSU, not file SYSTEM. If we are preparing a new system file, this bit will in any case be zeroed by the dead-start loader as it copies the system tape to file SYSTEM and sets up a new CMR directory.

Now we still have not put into the directory model the program name table entry for the new program, and the entry point table entries if it is a CP program. Subroutine PRIG, called by subroutine MAKE as it was formatting the program record, has saved the number of entry points in cell EPLIST, and their names in cells EPLIST+1 ff. So 2+ the number if EPLIST is the number of table entry words that have to be added to the directory model, and we put this number in X0. Cell PNT points to a word that stands between the entry point table and the program name table. This word in turn points to the end+1 of the program name table. We increase the latter pointer by X0, as all the additional words will be inserted at or below the end of the program name table.

Now if DESTIN does not contain 1, our model directory does not involve program bodies and their CM-addresses, so we avoid the next complication by going straight to ADDEC. But if DESTIN contains 1, we go to ADDEB, where we increase by X0 the CM addresses in all entries for CM-resident programs in the model program name table; because all those bodies will be moved up X0 words to make room for the new table words. Then, at ADDEA, do the same for the CM address in PROG+2 if the new program is CM-resident; its body is already in the model directory and will be moved up along with the rest, but its program name table entry is not yet in the table; we are still preparing the entry at PROG+1 and PROG+2. Then go to ADDEC.

At ADDEC, we now move all the bodies, if any, in the directory model up X0 words. Then at ADDED, increase the pointers in BEND and BOD by X0. They point to the end+1 of the whole directory model, and the end+1 of the program name table. Then put the second and third words of the new prefix, which constitute the program name table entry for the new program, in what are now the last two cells of the space for the program name table, just before the bodies, if any. Now if EPLIST contains 0, showing that there are no entry points, everything is in order, and we go to ADD1.

Otherwise, go to ADDF to start inserting entry points in the model entry point table. There is the correct number of free words in the model, but they are located between what was formerly the end of the program name table, and the program name table entry for the new program. So

we move up all the old part of the program name table, so that the vacant space comes at the end of the entry point table. At ADDG, we insert the program number of the new program after each entry point name in cells EPLIST+1 ff. Then if there is only one entry point, go straight to ADDH; otherwise, at ADDGA ff., sort them into alpahbetical order.

At ADDH, we merge the new entry points beginning at EPLIST+1 with the existing entry point table, causing the table to expand and fill the additional space made vacant for it. At the beginning of this process we add the number of entry points to the pointer in cell PNT, which is supposed to point to the word in the directory model that follows the end of the entry point table and procedes the beginning of the program name table. The word PNT points to is supposed to contain the address of the last+1 word of the program name table; we have already (just after ADDE) increased that address in that word by the number of entry points+2, and the word itself was moved up along with the program name table, when we made room at the end of the entry point table, at ADDF. Now we also add the number of entry points to the address in the first word of the directory model; this word is supposed to point to the same word PNT points to, although their contents actually differ by the amout we maintain in cell DIFF, i.e. the difference between the address relative to RA of our directory model, and the absolute address of the directory itself. Having merged in the new entry points, go to ADDU.

At ADDU, we have dealt with everything involved in adding the new program to the directory model except the necessity of sorting the program name table into the correct order. This will be taken care of by the COMPLETE function, after all the add's have been done. So now we check cell UPTO. If it contains zero, we have finished with this ADD, ADDBCD, ADDCOS, or ADDTEXT function and return to START. Otherwise, return to ADDB to get the next program from the source file, still using the same parameters that were decoded from the function card in the routine between ADD and ADDB.

Cell UPTO will contain zero if the ADD function card called for adding only one program, or if it called for adding more than one, and subroutine MAKE, in processing the last program from the source file, found that its name matched the terminal name in UPTO; then MAKE zeroed UPTO. If UPTO contained -1, this was a signal that we were to go to the end of the source file; when subroutine MAKE finds an end of file under this condition, it returns to START immediately because the ADD routine need not go any further.

DELETE routine, starting at DEL

First call subroutine MNA, to ensure that the first parameter (second element in the function card) is a name, and to move it to cell DELA. Now check cell DESTIN. If it contains 0, we are not between a READY and a COMPLETE, and DELETE must simply delete the named program from the running system. If so, call subroutine GETD to copy the running directory from CMR to out directory model space, and go to DELD.

If DESTIN does not contain 0, we are already preparing a directory model of some sort, and DELETE has to delete the named program from it. So go straight to DELD.

At DELD, call subroutine LOC to look in the model program name table for the program whose name is in DELA, and to put the address of the first word of its program name table entry in B2 and cell HOLD, and its program number in B4 and cell HOLD+1. Now look at the residence code in the program name table entry. If it is for CM-residence, and if DESTIN contains 0 or 1, we are dealing with the actual or potential running directory, and must call subroutine SQB to delete the body of the program from the directory model. If not, we do not call SQB. Then go to DELE. SQB, if called, identifies the program whose body is to be removed by the program name table entry addressed in B2.

At DELE, we reset B4 from cell HOLD+1, to contain the program number of the program to be removed, and then call subroutine SQE to remove its program name table entry, and its entry point table entries if any, from the directory model, and to adjust program numbers in the entry point table accordingly.

Then we check DESTIN again. If it does not contain 0, we are between READY and COMPLETE, and the model directory is to remain in our field length until the next COMPLETE card. So we go back to START. But if DESTIN contains 0, DELETE was called to remove one program from the running directory; and we call subroutine RTRN to copy the directory model back into CMR to replace the running directory. Then return to START.

Subroutines

We shall list the subroutines alphabetically for convenience.

CKNAME

This is called at various points by subroutine MAKE, to check a program name read from the source file, and to put it in PROG+1 as part of the prefix of the new program record.

The name is in X6 on entry, and is immediately stored in PROG+1. Then we check cell ONEDONE. If this contains zero, this is the first program to be handled by the present ADD, ADDBCD, ADDCOS, or ADDTEXT function card, and its name must match a parameter from the function card that has been stored in cell DELA. So we compare the name with cell DELA, and if they do not match, abort with an error message. If they do match, go to CKNMB.

If ONEDONE does not contain 0, either this is not the first program handled for the current ADD etc. function card, or the function card told us to begin at the current position of the source file without regard to the name of the first program found. So we do not check against DELA, and go to CKNMA. Now UPTO will contain either -1, indicating that the ADD etc. function is to continue to an end of file, in which case the current name cannot be the terminator, or UPTO will contain the name of the last program that is to be taken from the source file. So we match the new name against UPTO, and if they are the same, we zero UPTO to show that when the ADD routine has finished with this program, it will have completed the function card. Then go to CKNMB.

At CKNMB we are to make sure the new name does not duplicate a program name already in the model directory. But if cell COSFLAG is non-zero, we are doing an ADDCOS for "STITCH" program; they do not get such a check, and we exit from CKNAME immediately. Otherwise, beginning at CKNMD, match the new name against all those in the model program name table. If we find any matching entry that is not for a "STITCH" program, abort with an error message; otherwise exit from CKNAME.

Entry Information

The program name taken from the input record, in X6. Flags in ONEDONE, UPTO, COSFLAG. The current model program name table.

Exit Information

The program name has been stored in PROG+1.

Subroutines Called

None unless MSGB before aborting.

Registers Destroyed

A1, A2, A3, A6, X0, X1, X2, X3, X4, X6.

COPY

This is called by the TRANSFER, ADDTEXT, and COMPLETE routines to copy a record from a file being read through FET CBUF to a file being written through FET DBUF. It is assumed that the pointers in FET CBUF were initialized, and then as much as possible of the beginning of the record was read, after which the pointers and the status in CBUF were left undisturbed.

When COPY is called, X4 always = 0. If it did not, the record would also be copied into central memory beginning at the address in X4. But this feature is nowhere used in EDITLIB now, and we disregard it here.

At COPYD, we call subroutine EXC to copy the IN and OUT pointers from FET CBUF to FET DBUF, and to re-initialize the pointers in CBUF. It is assumed for COPY that the FIRST and LIMIT pointers in the two FET's are the same on entry, otherwise the re-initialization of CBUF would be disastrous.

Now we call subroutine MCBUF to write FET DBUF. If the buffer contained an integral number of PRU's, this will empty the buffer. If it does not contain an integral number of PRU's, the status of FET CBUF after the preceding read must have been e.o.r., and subroutine EXC has not disturbed this. So what is left over in the buffer as shown by the DBUF pointers will not be lost, as the next call to subroutine READ will see this status and exit immediately without reading further, and we will then write end of record from DBUF to clear out the buffer.

(One may object that the two files may use media with differing PRU lengths. But the buffer is 1025 words long. So as long as the media are either tape or disk, it is impossible to read, starting from an empty buffer, in such a way that neither are 1024 words read, nor is the CBUF status e.o.r. and as long as there are 1024 words in the buffer, each write we do will write all of them to either disk or tape.)

Having called MCBUF to write as much as possible, we call subroutine READ to read on FET CBUF. If the status is already e.o.f., the status return is a negative X1, and we go to TRAE to abort, because we have not seen the e.o.r. yet and something is wrong. If the status is already e.o.r., READ does not read, but exits with a positive non-zero X1. If so, we call subroutine MCBUF to write the end-of-record. FET DBUF has not been touched since our previous attempt to write. But if subroutine READ finds the status is not e.o.f. or e.o.r., our previous attempt must have emptied the buffer. The pointers in CBUF have been initialized by our last call on EXC, and READ will read 1024 words, or the rest of the record, and exit with X1 = 0.

On getting the zero response from READ, we return to COPYD above.

Entry Information

FET's CBUF and DBUF both refer to the same 1025-word buffer. FET CBUF has already been set to empty buffer, and as much as possible has been read into it (by subroutine READ in fact). The name of the output file is in DBUF. X4,0.

Exit Information

None.

Subroutines Called

EXC, MCBUF, READ.

Registers Destroyed

A1, A2, A3, A6, X1, X2, X3, X6.

DOCTS

This subroutine is called several times at the beginning and end of EDITLIB to bring to this control point, or make common if now local, or release from this control point (being common) the files SYSTEM, SSSSSST, and SSSSSSU. The calling sequence is:

```
+   RJ   DOCTS
-   EQ   X
```

where x is the address of a cell containing the name of the file, left justified with 0 fill in bits 18-59, and 0 in bits 0-17 if we are trying to get the file, or 4 if we are trying to release it.

We put the word x points to in cell CTSHOLD, and then call PP program CTS to work on that cell, with recall. The PP program leaves a response of 1, 3, 5, 7, 11B or 13B in bits 0-17 of CTSHOLD, and we exit from subroutine DOCTS with bits 0-3 of the response in.

For the meaning of responses when bits 0-17 of the request word were 0, see the comments in the assembly listing at the beginning of subroutine DOCTS. When we call DOCTS with bits 0-17 of the request word containing 4, we are trying to release the common file, and the response is 1 unless the file was not at our control point, which is impossible.

Entry Information

Only the calling sequence.

Exit Information

The response in X1.

Programs Called

CPC, CTS.

Registers Destroyed

A1, A6, X6.

EXC

This subroutine is called by the COPY subroutine and the COMPLETE routine, to facilitate copying a record. It copies the IN and OUT pointers from FET CBUF into FET DBUF and then resets IN=OUT=FIRST in FET CBUF. It is assumed that both FET's refer to the same buffer (BUF in fact). CBUF is being used for reading, and DBUF for writing, and EXC is called after reading and before writing.

Entry and Exit Information

Only the FET's.

Registers Destroyed

A1, A2, A3, A6, X1, X2, X3, X6.

GETD

This subroutine calls PP program MDI to copy the running system directory from CMR to our field length, beginning at the cell to which DIRPTR points. It is called by the MOVE, READY, DELETE, and LIST routines, and also at the beginning of EDITLIB, if the control card was not EDITLIB(RESTORE), to fetch the directory so that it can be saved on file SSSSSST.

In cells REQ and REQ+1, we set up a request for PP program MDI:

```
VFD   60/0
VFD   60/x
```

where x is the address in DIRPTR, showing where the directory model is to begin. Then call subroutine MDICALL, to call MDI.

Now the response in the same two cells is:

```
VFD   30/b,30/1
VFD   30/k,30/x
```

where x is still the same, k is the length of the directory, and b is the absolute address at which it begins in CMR. Into cell DIFF we put the difference between x and b, which we shall often use (though it is not explicitly mentioned in the narrative) when we have to make up pointers that will go into a model directory. We know where something is in the model, but the pointer to it in the model itself must point not to the cell something occupies now, but to the cell it would occupy if the model became the running directory. So in such a case we will subtract the content of DIFF from the address to produce the address that must be put in the directory; and vice versa on occasion.

The first word of the directory contains only a pointer to the word between the entry point table and the program name table. We put the address of that word, i.e. its address in our model, in cell PNT. That word, in turn, contains the address of the last+1 word of the program name table; we put into cell BOD that address as it relates to our model. It is also the address of the first word of a CM-resident program body. Then we find the last+1 address of the whole directory model, and save it in cell BEND. Then exit from GETD.

Entry Information

The pointer in DIRPTR, to our directory model area.

Exit Information

The copy of the directory in our field length, and cells DIFF, PNT, BOD and BEND as explained above.

Subroutines Called

MDICALL.

Registers Destroyed

A1, A2, A5, A6, X1, X2, X5, X6, B2.

GROW

This is called only once, in the LIST routine, to bring the entry point table and program name table of some system file, not of the running system, into our field length so that we can list the programs and entry points in the file.

We set X3 to point to the second word of our directory model area, and then call subroutine GROWA to read the next record of the file named in FET CBUF into the area beginning there. This file is presumably a system file. and has already been positioned at the beginning of the entry point table (immediately following DSD), by the LIST routine. On exit from GROWS, X3 points to

the first unused cell. We store this address in PNT, since in for a well-formed directory model, PNT is supposed to point to the word between the entry point table and the following program name table. Then increase X3 by 1, to skip over that word, and call subroutine GROWA again to read the next record of the file, which should be its program name table, into the next section of our directory model. On return, again, X3 has the address of the first unused cell. We put this in cell BOD, which, for a well-formed directory model, should point to the last+1 word of the program name table.

In order to make this directory model look as if we had copied it from the running directory, we must fix up the first word, and the word between the entry point table and program name table. But in order to set those pointer words correctly, we should know the real absolute address at which the CMR directory begins. So we format the request

```
VFD   12/3,48/0
VFD   60/anything
```

in cells REQ and REQ+1, and call subroutine MDICALL to call PP program MDI to carry out this request. All it does is return the absolute address of the CMR directory to bits 30-59 of cell REQ. This enables us to find the right value to put in cell DIFF, as well as to set up the pointers in the first word of the directory model, and the word between the entry point table and the program name table.

What is described in the preceding paragraph is really needless as we could just as well have set DIFF to contain zero and gone ahead on that basis. In turn, the capacity of PP program MDI to respond to this "type 3" request is unnecessary, as this is the only place in which it is used.

Earlier in the development of EDITLIB, it was supposed that one might bring in a directory from a system file by means of subroutine GROW, and after modifying it use it to replace the running directory. The waste motion in GROW has to do with that possibility, which was later eliminated.

Entry Information

FET CBUF must be set up to handle the relevant file, and the file must be positioned at the beginning of the entry point table.

Exit Information

What looks like a directory model.

Subroutines Called

GROWA, MDICALL.

Registers Destroyed

A1, A2, A3, A5, A6, X1, X2, X3, X5, X6, B6, B7.

GROWA

This is called only by subroutine GROW, at two points, to read the next record from the file named in FET CBUF into the area beginning at the address in X3, and to leave X3 pointing to the first unused cell.

First, call subroutine READ to begin reading the file named in FET CBUF, and to leave B6 and B7 containing the OUT and IN pointers from the FET after the read. On return from READ, X1 negative means that an end-of-file was read; so we abort with a message. A positive non-zero means that an end-of-record was read on the preceding call to READ, so that if we copied everything out of the buffer following that call, we are now through with the record, and exit from GROWA. If X1 contains zero, new information has been read from the record into the buffer, and B6 and B7 point to the first and last+1 words of this information. So we now copy this information word by word to where X3 points, increasing X3 by 1 after each word, and return to GROWB to continue.

Entry Information

The file must be correctly positioned, and FET CBUF must be set up for reading it. Bit 0 of word CBUF must be 1, and bit 4 must be 0 (i.e. no e.o.r. status X3 points to the first cell into which to copy).

Exit Information

FET CBUF is left with status = 03B; the file has been read one record forward; X3 points to the cell next after that into which the last word of the record was copied.

Subroutines Called

READ.

Registers Destroyed

A1, A6, X1, X6, B6, B7.

LIST

This subroutine (it is not the LIST function routine, which begins at LIST) is called whenever something is to be written on the OUTPUT file. The calling sequence is:

```
RJ     LIST
VFD    30/a,30/b
```

meaning, that from a through b-1 is a line to be listed. The line must begin with the appropriate format character (1 or blank).

We merely pick up the parameter word from the calling sequence and put it in the sixth word of FET OUTPUT, to indicate the workspace; then call subroutine IOWRITE (in program IO) to write it out.

Entry Information

Only the calling sequence.

Exit Information

None.

Subroutines Called

IOWRITE.

Registers Destroyed

A1, A2, A6, X1, X2, X6.

LOC

This subroutine is called by the DELETE and MOVE routines to search the model program name table for the entry for a program whose name is in cell DELA.

The directory model is assumed to have been correctly set up, along with pointers DIFF, PNT, BOD, and BEND. We get the pointers to the beginning and end+1 of the program name table, and put them in B2 and B3. Then initialize B4 at 0, as a program number counter. Then scan the program name table for a match to the name in DELA, increasing B4 by 1 after each P.N.T. entry has been rejected. If no match in the whole table, we give an error message and return to START. But if there is a match, B2 points to the P.N.T. entry, and B4 contains the program number. We leave these registers unchanged, but store them in HOLD and HOLD+1, and then exit.

Entry Information

A directory model, and a program name in cell DELA.

Exit Information

The address of the program name table entry with the same name, in B2 and HOLD; its program number in B4 and HOLD+1.

Subroutines Called

None.

Registers Destroyed

A1, A2, A6, X0, X1, X2, X3, X6, B3.

MAKE

This subroutine is called at two points in the ADD routine to read a record from a source file and format it as a system file record in the area beginning at PROG, with overflow on file SSSSSSX if the new record is over 4003B words long. If the record is a CP program, MAKE stores the number of entry points in EPLIST, and their names in EPLIST+1 ff. Otherwise, EPLIST is set = 0.

First we zero EPLIST. Then get the name of the SOURCE file from DELA+2. If it is "SYSTEM", go to MAKX to locate the input record from the model program name table. Otherwise, call subroutine PCBUF to prepare FET CBUF to read the source file. Now if DELA+4 contains non-zero modify CBUF to

read BCD instead of binary. Now we are at MAKA. Call subroutine READ to read as much as possible of the input record into the CBUF buffer (BUF ff.) and return with the response in X1, the starting address of the information read in B6, and its end+1 address in B7. If the response is non-zero, we can assume it is negative meaning an end-of-file was read. (PCBUF set the FET status to non-e.o.r., and if READ then read an e.o.r., the response would not show it until the following call to READ.) If e.o.f., we go to MAKB. This can only be legal if the function card was one of the two types:

```
ADD(x-*,f,r,e)
ADD(*,f,r,e)
```

and if ONEDONE is non-zero, showing that at least one program has already been assembled for this ADD function card. For either of those two formats, UPTO was initially set to -1. So now unless UPTO contains -1 and ONEDONE is non-zero, we abort with a message. Otherwise, we need not go back from the MAKE subroutine to the ADD routine, but can return straight to START.

If the response from the call to READ at MAKA was zero, we see whether the first word of the record begins with 7700B. If so, we assume it has a prefix, and go to MAKG. If not, we must be doing either an ADDCOS, and reading a Chippewa 1.1 CP program, or something of the ADDBCD/ADDTEXT type, and reading a BCD record beginning with a simple name card. First we test cell COSFLAG; if it is non-zero, we are doing ADDCOS; otherwise go to MAKI. If ADDCOS, the first word of the record should be the name; we pass it through subroutine CKNAME,which checks it for plausibility and stores it at PROG+1;then go to MAKL with X6=4, the type number for "STITCH" programs.

If we come to MAKI, hoping for ADDBCD/ADDTEXT action, the first word of the record should also be the program name, but it will have blank and zero fill on its right, instead of zero fill. So between MAKAT and MAKAW we check that this word begins with plausible characters, possibly followed by blanks, possibly followed by zero bytes. (Note that we do not check the length of the name; however, subroutine CKNAME will curtail it to 7 characters.) Then call subroutine CKNAME to check the name for duplication and store it in PROG+1. Now as we are going to turn the record into an overlay, and we do not want the first word of its body to be its name any more, we replace the first word in the read buffer, which was the name, by 50000101000000000000B, which is proper as the first body word of an overlay. Then go to MAKL with X6=2, the type number for an overlay.

Let us return to the case when the input record does begin with a prefix. We come to MAKG, extract the prefix length from its first word, and store the length at PLENG. Then see if the first word of the prefix ends with 7777B. If so, the record is already formatted for system files, as only EDITLIB writes this kind of prefix. If so, see if it is type 4 ("STITCH") and if not of that type, go to MAKGH. If "STITCH", do not use subroutine CKNAME, as we need not check for name duplications, but merely store the name at PROG+1 and go to MAKGL. But if no "STITCH", go to MAKGH as if the record were not already formatted. At MAKGH, call subroutine CKNAME to check the record name for duplication and store it at PROG+1.

If the record was not already formatted in system file form, we also went to MAKGH to call CKNAME. So after calling CKNAME, we again check if it was formatted in system file form. If so, go to MAKGL; if not, to MAKK.

We come to MAKGL if the record name is acceptable, and the record is already in system file form. We extract the body length from the second word of the prefix. But if this is zero, the record was not already formatted after all (it is conceivable that some program might produce a binary record beginning with a prefix whose first word ended in 7777B, but surely the name would be in the second word with bits 0-17=0. Only a system-formatted record would have those bits utilized for the body length) and we go to MAKK. Otherwise, store the third word of the record at PROG+2, extract its type code and save it at EPLIST, call subroutine PRIG to copy the rest of the input record into PROG+3 ff. (using file SSSSSSX for overflow if necessary)and continue at MAKGM. This is the only way of arriving at MAKGM.

Between MAKGM and MAKGA we consider the following: For the residence code and edition number, we must use, in descending order of preference,

a. codes explicitly given on the function card.

b. codes taken from the running directory, if the program is copied from the running system. These, or at least the residence code, might not be the same as that in the prefix of the record itself, if a "MOVE" had been done previously.

c. codes taken from the prefix of the record, if it is already formatted as a system file record.

d. residence 1 for disk, and edition number 0.

So if we have just taken a record from "SYSTEM", we replace the residence and edition codes from the program name table and insert them in the prefix of the record at PROG+2, so as to enforce the preference of b. over c. above.

Between MAKGA and MAKFA, in turn, if there were explicit codes on the function card (non-negative numbers now in DELA+3 and/or REDAC), we insert them in the prefix in PROG+2 to enforce the preference of a. over b. above. Then rejoin, at MAKFA, the sequence that begins at MAKK.

At MAKK, we start dealing with a record that is not already formatted as a system file record. By the prefix length saved in PLENG, advance the OUT pointer in B6 so as to skip the prefix, which has no further interest. Now we must decide what kind of program this is, according to the first word of its body. If this word begins with 5000B, it must be an overlay, and we go to MAKL with 2 in X6, as the type number. If the word begins with 3400B, we must have a CP program, and we go to MAKL with 1 in X6. Otherwise, go to MAKL with 0 in X6, the type number for a PP program. The body of a PP program begins with its name, which must have 3 characters, so it could not possibly start with 5000B or 3400B.

We may also have come to MAKL from a point further back, with 4 in X6 because we are doing ADDCOS, or 2 in X6 because we are turning a BCD record into an overlay. In any case, the record is not already formatted as a system file record, so the choices for residence and edition number codes

are only a. and d., as explained in the third paragraph back, and if explicit codes were given on the function card, as indicated by non-negative numbers in DELA+3 and/or REDAC, we insert them in PROG+2; otherwise use 1 for residence and 0 for edition number. Then call subroutine PRIG to copy the rest of the input record into PROG+3 ff., using file SSSSSSX as overflow if necessary, and go to MAKFA.

When we get to MAKFA, we have set up the properly-formatted program record in PROG ff., using file SSSSSSX as overflow if necessary, and subroutine PRIG has extracted the entry point names, if any, putting their count in EPLIST and storing their names at EPLIST+1 ff. Now, if ONEDONE contains 0, we set it to 1 to show that at least one program has been handled for this ADD function card. This means that an end-of-file may be acceptable on the source file the next time MAKE is called, and that the name of the next record on the source file need not be checked with the first program name on the function card. If ONEDONE is already non-zero, we carefully leave it alone, because it may contain a pointer to an entry in the program name table at SDIR ff.

Then we exit from subroutine MAKE.

Entry Information

DELA+4 contains non-zero if the source file is to be read in BCD. DELA+2 contains the name of the source file, or "SYSTEM" (which means the record may actually be on SYSTEM or on SSSSSSU).

DELA contains the name of the first program mentioned on the function card.

ONEDONE contains 0 unless either MAKE has already been called during the execution of this function card, or the card was of the type "ADD(*SYSTEM, res,edn)" in which latter case ONEDONE points to the cell 2 below the next model program name table entry to be used.

UPTO contains 0 if the function card called for a single program to be added, or the name of the last program to be added, or -1 if the function card called for continuing to the end of the source file.

DELA+3 contains the numerical residence code translated from the function card, or a negative number if this was absent.

REDAC contains the edition number translated from the function card, or a negative number if this was absent.

Exit Information

ONEDONE is non-zero. If the source is "SYSTEM", it contains the address of the entry in the model program name table in SDIR ff., that was just used.

PROG ff. contains the new record, that the ADD routine will now write out on file SSSSSSS or SSSSSSU.

SXCT contains either 0, in which case the new record is entirely in PROG ff., or some number k such that the first three words of the record are in PROG through PROG+2; the next k*4000B are at the beginning of file SSSSSSX, and the remainder are in PROG+3 ff. The size of the remainder can be found from the total length of the program, -3, contained in its prefix in PROG+1. FET DBUF is set up so that writing end-of-record on it will send the remainder to file SSSSSSX once the IN pointer is correctly set.

Subroutines Called

PCBUF, CKNAME, READ, PRIG.

Registers Destroyed

All but A0, A4, A5, X5, B1.

MDICALL

MCBUF

This is called at many places in EDITLIB to read, read-skip, write, write e.o.r., write e.o.f., backspace, or rewind the file named in FET CBUF or DBUF. The calling sequence is

```
+   RJ   MCBUF
-   JP   f
```

if FET CBUF is to be used, or

```
+   RJ   MCBUF
-   EQ   f
```

if FET DBUF is to be used; f is the appropriate function code with the mode bit 0 whatever the mode of the file; i.e. f will be 10B, 20B, 14B, 24B, 34B, 40B, or 50B.

We isolate the function code from the calling sequence and combine it with the name "CIO" and a recall bit into a request word which we store in MCBUFB. Then set A1 to point to the proper FET, according to the calling sequence, and call subroutine CPC to format and issue the PP request.

Entry Information

The calling sequence and the pointers in the FET to which it refers.

Exit Information

None.

Subroutines Called

CPC (in program CPC).

Registers Destroyed

A1, A2, A6, X1, X2, X6.

MCBUFC

This subroutine is called whenever a request for PP program MDI has been set up in cells REQ and REQ+1, to call MDI. There is not much point to this subroutine; the checking for completion that it does is unnecessary. At a certain stage in developing EDITLIB, MDI had to be elaborately simulated, and it was worthwhile then to have a subroutine.

Entry Information

The request in REQ and REQ+1.

Exit Information

The response in REQ and REQ+1.

Subroutines Called

CPC (in program CPC).

Registers Destroyed

A1, A6, X1, X6.

MEM

This is called at the beginning of EDITLIB to make sure there is room in our field length for a directory model 30000B words long. Later it may be called by the LENGTH routine to expand the field length. The calling sequence is

```
RJ      MEM
DATA    x
```

where x is the maximum length for a directory model. Then the field length is to equal at least x+y+100B, where y is the starting address for directory models, given in DIRPTR. The present field length, minus 100B, is stored at BENDLIN. If this is already enough, exit from MEM without doing anything. Otherwise, set up a request in cell HOLD for the required amount and then call PP program MEM through subroutine CPC. Also update BENDLIN.

Entry Information

The calling sequence, DIRPTR, BENDLIN.

Exit Information

BENDLIN updated.

Subroutines Called

CPC (in program CPC).

Registers Destroyed

A1, A6, X1, X6.

MSGB

This is not a complete subroutine, but a variant entry for subroutine LIST. As for LIST, the calling sequence is

```
RJ      MSGB
VFD     30/a,30/b
```

where a message extending from a through b-1 is to be written as one line on the OUTPUT file. But this is to be preceded by a listing of the current function card. So first we call LIST from MSGB, to list the function card

with a blank word before it for the sake of the format character. Then simply move the entry word from MSGB to LIST and branch to LIST+1, as if LIST rather than MSGB had been called in the first place.

Entry Information

Only the calling sequence.

Exit Information

None.

Subroutine Called

LIST

Registers Destroyed

A1, A2, A6, X1, X2, X6.

PCBUF

This is called at various points, with a file name in X1, to prepare FET CBUF for reading the file into buffer BUF. If bits 0-17 of X1 are zero, the mode is set to binary. Otherwise, the mode is set according to bit 1 of X1. We put the file name in bits 18-59 of cell CBUF, 000001B or 000003B (according to mode) in FIRST pointer. It is assumed that the FIRST pointer always points to BUF, and the LIMIT pointer to BUF+2001B.

Entry Information

A file name, possibly along with a status code, in X1.

Exit Information

None.

Subroutines Called

None.

Registers Destroyed

A6, A7, X1, X6, X7.

PRIG

This is called at two points in subroutine MAKE to copy the rest of the input record into PROG+3ff., after the first three words of the new program record have been formatted in PROG through PROG+2. B6 points to the last word of the prefix of the input record (counting a BCD record whose first word is its name as having a one-word prefix).

First we set FET DBUF to write on file SSSSSSX using the area between PROG+3 and PROG+4002B as its buffer. We shall need this only if the remainder of the input record turns out to be more than 3777B words. Now we zero SXCT, the count of how many times we have written 4000B words on file SSSSSSX, and call subroutine MCBUF to rewind the file. If the file does not already exist, this creates it, and in any case we fortunately do not care whether it existed before, or was common or local, as long as we can write on it.

Next we set B2 to PROG+3, the address at which to start storing the body of the output record. Now check the program type number at EPLIST. If this is 1, go to the more complicated section of the subroutine at PRAG, because this is a CP program and we have to extract and list the entry points. Otherwise, merely zero EPLIST, which now becomes the count of entry points. This brings us to PRIGA. Between PRIGA and PRIGC we copy words from the input buffer (B6 contains the OUT pointer and B7 the IN pointer, in effect) to the output buffer; putting a word in the output buffer is done simply by calling subroutine PRUG with the word in X6. When B6=B7, call subroutine READ to re-fill the input buffer, and to reset B6 and B7. When the response from READ, in X1, is 0, we continue processing. If the response is negative, we have struck an end-of-file and abort with a message. If the response is positive non-zero, we reached an end-of-record during the preceding call to READ; having transferred all the words that were read by that call, we have now completed the record and go to PRIGC.

At PRIGC, we check DELA+4. If this is zero, go straight to PRIGCA. Otherwise, we must be doing an ADDTEXT, ADDBCD, or ADD with minus as the first parameter. For the particular type of overlay this is to make up, the ultimate user of the overlay wants the last word to be 77777777777777777777B, so we now pass such a word through subroutine PRUG to add it to the end of the output record. Then go to PRIGCA.

At PRIGCA, we find the length of the body of the output record (i.e. the length of the whole record minus 3 for the prefix) and put it in bits 0-17 of PROG+1; then exit from PRIG. The length is found by subtracting PROG+3, the starting address of the body, from the content of B2, which points to the next unused word in the buffer, and adding 4000B times the content of SXCT, which gives the number of words that have been written out on file SSSSSSX.

We come to PRAG when we find we have to deal with a CP program and its entry points. First set B3 to EPLIST+1, where the list of entry point names in the program is to begin. The program record must be divided into "tables", and at PRAGF begins the cycle for dealing with each table.

At PRAGF, if the input buffer is not exhausted, go to PRAGA. If it is exhausted, call subroutine READ; now if the response is negative, we have struck end-of-file, and abort with a message. If the response is positive non-zero, the last READ call, whose harvest we have already processed, gave an end of record, and we go to PRAGB to store the number of entry points in EPLIST and go to PRIGC. Actually we might as well go to PRIGCA at this point. There we put the length of the body of the new record in its prefix, and exit from PRIG.

But if the response from READ is zero, B6 and B7 have been reset, and we go on to PRAGA. Here we pick up what must be the first word of a table; save its length-1 in B4, and set X3=0, if and only if the word begins with 3600B, indicating that this is a table of entry points. Now, at PRAGE, call subroutine PRUG to store the word in the output buffer. Then if B4 = 0, the table is exhausted and we go back to PRAGF to begin the next table, or end the program. Otherwise, we get the next word of the input record, and count down on B4. (If the input buffer is exhausted, we call READ; either an e.o.f. or an e.o.r. response will cause an abort here, because it is intolerable for the record to end in the middle of a table.) If X3 is not = 0, we are not in an entry point table and can merely go back to PRAGE to store the word and pass on. If X3 = 0, the word may be either an entry point name, or the corresponding value. If bits 54-59 of the word are 0, it is the value, and we merely go back to PRAGE. Otherwise, it is the name, and we store it at B3 and step B3. Then go back to PRAGE to store the word in the output buffer as well.

Entry Information

PROG, PROG+1, PROG+2 are already set up. FET CBUF is set up for reading the input file; subroutine READ has been called once to start reading this record, and the FET has not been touched since then. B7 still contains the IN pointer from that FET, and B6 is our effective OUT pointer, pointing to the last word of the prefix of the input record. EPLIST contains the type number of the record - 0 for PP program, 1 for CP program, 2 for overlay, 4 for "STITCH" program.

Exit Information

The number of entry points is in EPLIST, and they are stored in EPLIST+1 ff. The length of the program record body has been filled into bits 0-17 of the second word of its prefix, at PROG+1. If SXCT contains 0, the output program record is complete in PROG ff. Otherwise, SXCT contains a number k such that the first three words of the record are in PROG through PROG+2; the next k*4000B are at the beginning of file SSSSSSX, and the remainder are in PROG+3 ff.; FET DBUF is set up so that writing end-of-record on it will send the remainder to file SSSSSSX once the IN pointer is correctly set.

Subroutines Called

READ, PRUG, MCBUF.

Registers Destroyed

A1, A2, A6, X1, X2, X3, X6, B2, B3, B4, B5, B6, B7.

PRUG

This is called only from several points in subroutine PRIG, to put the word in X6 into the next available position in buffer PROG, updating B2 accordingly, and to put the overflow on file SSSSSSX if necessary.

On entry, the address of the next available word in the buffer is in B2. We store the word there, and increase B2 by 1. If it now = PROG+4003B, the buffer is full; otherwise exit from PRUG.

If the buffer is full, we set the IN pointer of FET DBUF to PROG+4003B and call subroutine MCBUF to write on it. The FET has already been prepared so that this causes PROG+3 to PROG+4002B inclusive to be written as an integral number of PRU's on file SSSSSSX. Then we reset the IN and OUT pointers of the FET to PROG+3, add 1 to the content of SXCT, reset B2 to PROG+3, and exit from PRUG.

Entry Information

X6 contains a word to be stored in the PROG buffer; B2 contains the address of the next cell available for storage. FET DBUF is set up as to everything but its IN pointer to write from PROG+3 on, to file SSSSSSX.

Exit Information

B2 is updated. If the word just stored filled the buffer, SXCT has been increased by 1.

Subroutines Called

MCBUF

Registers Destroyed

A1, A2, A6, X1, X6, B5.

RDR

This subroutine is called at two points in the SKIPF routine to read the beginning of a record using FET CBUF, and then skip if necessary to the end of the record. On exit, X1 will contain 0 normally, or non-zero if an end-of-file was read.

We get the file name out of CBUF itself, and call subroutine PCBUF to reset the status to 1 or 3, and to set IN=OUT=FIRST in FET CBUF. Then call subroutine MCBUF to do a read-skip. Then check for end-of-file response in the FET and exit with the corresponding value in X1.

Entry Information

The file name and mode in CBUF.

Exit Information

FET CBUF as altered by the read-skip. X1 contains 0 unless an end-of-file was read.

Subroutines Called

MCBUF, PCBUF.

Registers Destroyed

A1, A2, A6, A7, X2, X6, X7.

READ

This is used to begin reading a record through FET CBUF, by the TRANSFER, ADDTEXT, and COMPLETE routines, and by subroutines COPY, MAKE, PRIG, and GROWA.

On entry, we see whether bit 4 of CBUF is 1, indicating a previous e.o.r. or e.o.f. status. If so, go to READF, zero that bit, and exit with X1=1. Otherwise, with the first word of FET CBUF already in X1, call subroutine PCBUF, which sets the FET status to 1 or 3, and sets the pointers IN=OUT= FIRST (=in fact BUF). Then call subroutine MCBUF to read the file. If the status afterwards is end-of-file, reset it to 1 or 3 and then exit from READ with X1 negative. Otherwise, go to READA. Set B7 to the IN pointer, and B6 to address BUF, which must be the value of the OUT pointer. Before exiting, see if B6=B7. If so, we must have just read a zero-length PRU that ends a record, and as there is no more information to pass back from READ, we might as well give the e.o.r. indication. So we go back to READ+1 as though we had returned to the calling routine with X1=0 and B6=B7, and the calling routine had simply called READ again.

Entry Information

The first word of FET CBUF.

Exit Information

B6 and B7 contain the addresses of the first and last+1 words of information in the buffer, with no circularity, if X1=0. If X1 is positive non-zero, there is no information in the buffer, and the FET status is end-of-file and we have done nothing with the pointers.

Subroutines Called

PCBUF, MCBUF.

Registers Destroyed

A1, A2, A6, A7, X0, X2, X6, X7.

RECALL

This subroutine is called several times near the beginning of EDITLIB, while we are waiting for a common file at some other control point to be released and become available. RECALL merely calls CPC to relay an "RCL" call to monitor, so as to drop the central processor while waiting for the common file.

Entry and Exit Information

None.

Subroutines Called

CPC (in program CPC).

Registers Destroyed

A1, A6, X1, X6.

RES

This is called by the MOVE and ADD routines to translate a symbolic residence code, taken from a function card, into a numerical code that is suitable for the four-bit field in the second word of a program name table entry.

We enter RES with the symbolic code in X1. If X1 contains zero, it means we have gone past the last parameter on the function card, and are now taking a "missing parameter" from the breakdown list. In that case we immediately exit from RES with a negative number in X6 for a response; the ADD routine will take this to mean "use the residence already in the record prefix, if it is formatted for a system file, and otherwise call it disk-resident".

Otherwise, we match the code in X1 with successive words in table RESA, stopping on a zero word in the table. If a match is found, exit with the ordinal of the matching table word in X6, i.e. 0 or 1. If no match go to MSGA to give a message and go back to START, thus abandoning this function card.

If and when other residences become available, their symbolic codes should be put in table RESA between "DS" and the terminal zero.

Entry Information

The symbolic residence code in X1.

Exit Information

Its numerical equivalent in X6, or negative X6 if X1 was initially 0 showing an absent parameter.

Subroutines Called

None.

Registers Destroyed

A2, X2, B2.

RTRN

This subroutine is called by the MOVE, DELETE, and COMPLETE routines to copy a directory model into central memory resident to replace the running directory. It is also near the beginning of EDITLIB if the control card is "EDITLIB(RESTORE)" to restore to CMR a directory that had earlier been saved on file SSSSSST, and has just been read back from it.

The model directory is assumed to be completely formatted, except that the word between the entry point table and the program name table has to have inserted in bits 24-41 the address of the first P.N.T. entry for a non-PP program, or 0 if by chance there are no non-PP programs. Bits 0-17 contain the address of the last+1 word of the program name table, and this we leave unchanged.

So we begin by locating the word between the entry point table and the program name table, and zeroing bits 18-59. Then we scan the model program name table from the beginning until we find an entry whose type code is not zero. If none such, go on to RTRNK, otherwise, put the address of the first one found, into bits 24-41 of that word. Note that this address is calculated artificially so that it will be correct once the model directory is transferred to CMR.

At RTRNK, we construct in cells REQ and REQ+1 a request for PP program MDIC

```
VFD 12/1,48/0
VFD 30/a,30/b
```

where b is the starting address of the directory model, taken from cell DIRPTR, and a is the length of the model, found by subtracting the content of DIRPTR from the content of BEND. Finally, call subroutine MDICALL, which calls PP program MDI to obey the request by replacing the running directory with this model. For information on how the SCOPE system is protected during this change of directory, see the notes to program MDI.

Entry Information

A well-formed directory in the space beginning where DIRPTR points, except for the one pointer whose insertion is mentioned above; the pointers in DIRPTR, PNT, BOD and BEND.

Exit Information

None.

Subroutines Called

MDICALL

Registers Destroyed

A1, A2, A6, A4, X0, X1, X2, X4, X5, X6, B3, B4.

SQA

This subroutine is called by subroutines SQB and SQE. It goes through the model program name table and subtracts the number in bits 0-17 of X0 from the CM-address in the entry for every CM-resident program entry. However, it does not touch such an address if already less than X5.

SQA is called by SQE when we have just removed some P.N.T. and E.P.T. entries from the model directory; as these lie below all bodies of CM - resident programs, all the addresses of those bodies have been effectively reduced. So we come into SQA with X0 = the number of table words removed, and X5 = 0 so that the whole program name table is covered.

SQA is called by SQB when we have just removed a body from the model directory. We enter SQA with the length of that body in X0 and its former starting address in X5, so that only the addresses of bodies that lay beyond it are reduced.

Subroutine SQA begins by shifting the number in X0 to correspond with the CM - address field in the words it may have to be subtracted from. Then check cell DESTIN. If it does not contain 0 or 1, we can exit from SQA without doing anything, because the model directory does not contain any bodies anyway. It must be for a new system file, and CM-addresses will be filled into the program name table as necessary if and when that file is dead-started.

At SQAT we get from PNT and BOD the pointers to the first-1 and last+1 words of the program name table. Then we scan the whole table; ignore non-CM-resident entries; and for each CM-resident reduce its CM-address by X0 if it is previously greater than X5. Then exit from SQA.

Entry Information

A well-formed directory model, with pointers in PNT, BOD and BEND. A file name in DESTIN if we are between a READY (file name) function and an expected COMPLETE function; or 1 in DESTIN if we are between READY (system) and COMPLETE or 0 in DESTIN if we are not between READ and COMPLETE at all. In the last case, we must be doing a MOVE or DELETE function for a single alteration of the running system. X0 contains a reduction constant, and X5 contains a threshold address below which a CM-address is to be exempt from reduction. Note that the address in X5 (except for the trivial case X5=0) is not the address of a body in the directory model, but the address the body would begin at if the model became the running directory.

Exit Information

None.

Subroutines Called

None.

Registers Destroyed

A1, A6, X0, X1, X2, X6, B4, B7.

SQB

This subroutine is called by the MOVE and DELETE routines to remove from the model directory the body of a program that is either being deleted altogether, or being degraded from CM-resident status, and to adjust as necessary the CM-addresses of other programs in the model program name table.

On entry to SQB, the address of the model program name table entry of the program being deleted or moved is in B2. First refer to that entry (if by chance it were not now for a CM-resident program, there would be a disaster) and put the CM-address of its body in X1 and the length of its body in B4. Then add the constant in KDIFF to X1 so as to get the address of the body in the model, rather than in the CMR directory. We

also zero the CM-address field in the program name table entry, so as to leave it free, if this is a MOVE rather than a DELETE, for the additional part of the disk address or eventually for the address in some other kind of storage. Now we subtract the length of the body to be removed from the pointer in cell BEND, which is supposed to be the address of the last+1 word of the directory model.

We have the starting address of the body to be removed in B3, its length in B4, and the new end+1 address of the directory model in B7. If B3 = B7, the body to be removed was the last thing in the model, so we need not do any shifting and exit from SQB. Otherwise, move everything between B3+B4 and B7+B4-1 back B4 words in memory. Now the pointers to all the bodies we have just moved back will have to be reduced by B4. We call subroutine SQA for this. It requires X0 to contain the number to subtract from the relevant CM-addresses in program name table entries, and X0 already = B4. SQA also requires that X5 contain a number which any such CM-address must be above in order to require reduction by X0. We already have in X5 the address as it was in the program name table entry, of the body we have just removed, and this is correct for SQA. (Note that X5 is the address of the body as it would be in the CMR running directory, not as it is in our model directory area, which is why we had to add the content of DIFF to X5 to get B3.) Then exit from SQB.

Entry Information

A well-structured directory model, with pointers in DIFF, PNT, BEND; the address of the relevant model program name table entry in B2.

Exit Information

The model and pointers have been updated.

Subroutines Called

SQA.

Registers Destroyed

A1, A2, A3, A6, all X, B3, B4, B6, B7.

SQE

This subroutine is called only by the DELETE routine, to remove from a directory model the program name table entry and entry point table entries for a program. On entry, B4 is the number of the program.

First we remove the program name table entry. We locate the word that follows this entry in the directory, and the last+1 word of the whole directory model. Then reduce the pointers in BEND and BOD (pointers to the end+1 of the whole directory, and to the end+1 of the program name table) by 2. Then, at SQEK, move everything above the P.N.T. entry to be removed, back two words.

Next we have to remove all the entry point table entries for this program. Also, because removing the program name table entry has effectively reduced by 1 all higher program numbers, we must reduce them accordingly in the entry point table. So we set pointers to the beginning and end+1

of the entry point table, and preset B5=0 a counter of entries removed. B4 is still the program number of the removed program. For each entry point table entry, if its program number = B4, we simply increase B5 by 1. If its program number is below B4, we replace it B5 cells lower than where we found it. If its program number is above B4, we reduce its program number by 1 and then replace it B5 cells lower than where we found it.

After this, if B5=0, no entry point table entries were removed and we go straight to SQEF. Otherwise, we reduce by B5 the pointers in PNT, BOD, and BEND, as we have removed B5 words from a section of the directory model which they all point beyond. Then, at SQED, move everything in the directory model that lies after the entry point table back B5 words.

We come to SQEF when we have removed 2 program name table words and B5 entry point table words from the directory model and shifted everything back to close up the gaps, and adjusted pointers PNT, BOD, and BEND, the word between the E.P.T. and P.N.T., and the first word of the directory. It remains to reduce by B5+2 the CM-addresses for all CM-resident programs in the program name table, since all bodies have been moved back this much. So we set X0=B5+2, as the reduction constant, and X5=0, as the number which a CM-address must exceed if it is to be reduced (i.e. all CM-addresses are to be reduced). Then call subroutine SQA to do the reduction, and then exit from subroutine SQE.

Entry Information

A well-structured directory model, with pointers in DIFF, PNT, BOD, and BEND; and a program number in B4.

Exit Information

The directory and pointers have been updated.

Subroutines Called

SQA.

Registers Destroyed

All but A0, A3, A4, A5, X3, X4, X5, B1.

SRB

This subroutine calls PP program SRB to convert a 24-bit disk record address, with the name of the file known, into a 45-bit adisk address that is more convenient for PP-resident to use on occasion, and that is sufficient even without knowing the name of the file.

It is called by the MOVE routine, which if moving a program from CM-resident to disk residence, must replace the short disk address and CM-address in its program name table entry by a long disk address. It is also called by the ADD routine when we are between READY(SYSTEM) and COMPLETE, when adding a disk-resident program to the directory model. In this case, we wrote the record on file SSSSSSU, and at that time got its 24-bit disk-address from the disk drivers; now we want the 45-bit disk address to put in the directory model before it goes back to CMR.

On entry to SRB, we have the file name in X1, the address of the cell that contains the 24-bit record address in A2, and that word itself in X2. We store the file name at SRBA, extract the 24-bit record address and store it at SRBA+1, and store the address of that word at SRBA+2. Then call subroutine CPC to call SRB, with a request pointing to SRBA. The PP program uses SRBA and SRBA+1, and leaves the 45-bit record address in SRBA+1. The address in SRBA+2 is only for the convenience of our CP program, which now inserts the 45-bit address in the word SRBA+2 points to; which is of course the second word of the relevant program name table entry. Then exit from SRB.

Entry Information

X1 contains the name of a file (left justified with zero fill); A2 points to the second word of a program name table entry; X2 contains that word.

Exit Information

Bits 24-44 of the second word of the program name table entry have been replaced by the extension of the disk record address in bits 0-23, for the file whose name was in X1 on entry.

Subroutines Called

CPC (in program CPC).

Registers Destroyed

A1, A2, A6, X1, X2, X6.

SRT

This subroutine is called by the MOVE and COMPLETE routines, to put the program name table in the model directory into proper sequence, and adjust program numbers in the entry point table accordingly. MOVE, COMPLETE, and DELETE are the only routines that can send a completed or altered directory back to the CMR or out on a new system file. DELETE does not need to call SRT, because either it comes between READY and COMPLETE, in which case COMPLETE calls SRT before the directory is considered finished; or else it merely deletes one program from the running directory, and deleting one program cannot disturb the proper mutual ordering of the rest. But MOVE needs SRT, because changing the residence of a program probably changes the proper position of its entry in the program name table.

Program name table entries must be sorted:

1. Primarily, on ascending record type number, in bits 52-55 of the second word of each program name table entry; viz., PP program, CP program, overlay, "STITCH" program.
2. Secondarily, on ascending residence code, in bits 56-59 of the same word; viz., CM-resident, then disk-resident.

We are going to sort program name table entries by the method of comparing each pair of neighbors in turn and exchanging them if their mutual order is incorrect. After repeatedly going through the table in this way, the

sort is complete when we find we have gone through the table without having to exchange any pair of neighbors. Before sorting, we set up a table of consecutive integers, one per entry, in cells SDIR ff. Whenever we exchange a pair of entries in the entry point table, we shall exchange the corresponding integers in the table at SDIR. When the entry point table has been completely sorted, the table of integers shows how the program numbers have been changed. If cell SDIR+x contains integer y, this means that what used to be program number y is now program number x.

Then we go through the model entry point table. From each word extract the old program number y. To find the new program number of the same program, we must search the table beginning at SDIR for a word containing integer y. This having been found at SDIR+x, we replace the program number in the entry point table entry by x. When this has been done for the whole entry table, we exit from SRT.

Entry Information

A well-formed directory model, with pointers in PNT, BOD, BEND.

Exit Information

The same model has been sorted.

Subroutines Called

None.

Registers Destroyed

All but A0, A3, A4, A5, B1, B7.

FLOW CHARTS

A decision point is always represented by a hexagon, with the two alternate exit lines appropriately labelled. A hexagon always represents a decision point.

A lozenge always represents a subroutine call. The name of the subroutine is given inside the lozenge, with no enclosing characters. A subroutine call may also be represented by the name of the subroutine framed by two slashes (e.g. /SRB/) inside a rectangle.

A rectangle may indicate anything but a decision point, but when a rectangle contains a subroutine call, the name of the subroutine is enclosed by slashes as stated above.

A symbol enclosed by asterisks (e.g. *PQR3*) corresponds to a location symbol in the program listing. Such a symbol may occur as the first line inside a hexagon, lozenge, or rectangle indicating that the action described by succeeding lines begins at the so-named location in the program listing, e.g.:

```
*PQR3*
LDN 5
/STS/
```

inside a rectangle would indicate that beginning at location PQR3, 5 is

put in the A-register and then subroutine STS is called.

A branch to a point on the same page, if it cannot be indicated by a line to that point, is indicated by a line to a rectangle containing only the name of the point, enclosed by asterisks.

A branch to a point on a different page is indicated by a line to a rectangle containing the name of the point, enclosed by asterisk, with the page number below it in parentheses. E. g.:

IDF
(PAGE 5)

Double parentheses are used as quotation marks, enclosing actual or imaginary literals.

Single parentheses are used often to mean "contents of". E. g. SET (D.Z1) = BUF would mean "set the contents of cell D.Z1 equal to the number whose symbol is BUF". But these parentheses are often omitted. E.g. D.Z4=0 inside a hexagon would mean "does cell D.Z4 contain zero".

Index to Location Symbols in EDITLIB Flowcharts.

NAME	PAGE	NAME	PAGE	NAME	PAGE
ADD	15	AT	14	ENDB	5
ADDA	16	ATA	14	ENDD	5
ADDAB	16	ATB	14	EXC	29
ADDAC	16	ATC	14	GETD	23
ADDAD	16	BKS	11	GROW	38
ADDADB	16	BKSA	11	GROWA	38
ADDADC	16	BKSB	11	GROWB	38
ADDAV	15	CKNAME	34	GROWD	38
ADDAW	15	CKNMA	34	KLUGA	4
ADDAY	15	CKNMB	34	KLUGE	4
ADDB	17	CKNMD	34	KLUGH	4
ADDBA	17	CKNMY	34	KUGEL	3
ADDBAC	17	COM	20	LASA	21
ADDBACH	17	COMA	20	LASB	21
ADDBACK	17	COMB	20	LASC	21
ADDBACL	18	COMF	20	LEN	5
ADDBCD	15	COMG	20	LIS	21
ADDBE	18	COMK	20	LISC	21
ADDC	17	COML	20	LISE	21
ADDCOS	15	COPY	29	LISQ	21
ADDE	19	COPYD	29	LIST	24
ADDEA	19	DEL	13	LOC	23
ADDEB	19	DELD	13	LOCB	23
ADDEC	19	DELE	13	MAKA	30
ADDED	19	DELK	7, 13	MAKAWA	31
ADDEF	18	DOCTS	22	MAKAWB	31
ADDF	19	EDITA	2	MAKB	30
ADDG	19	EDITB	2	MAKE	30
ADDGA	19	EDITE	1	MAKFA	32
ADDH	19	EDITLIB	1	MAKG	31
ADDHB	19	EDITM	3	MAKGA	33
ADDU	19	ENDA	5	MAKGB	33

Index to Location Symbols in EDITLIB Flowcharts (cont.)

NAME	PAGE	NAME	PAGE	NAME	PAGE
MAKGH	31	PRAGD	36	SQA	25
MAKGL	31	PRAGE	36	SQAB	25
MAKGM	33	PRAGF	36	SQAC	25
MAKI	31	PRIG	35	SQAT	25
MAKK	32	PRIGA	35	SQB	25
MAKL	32	PRIGB	35	SQE	26
MAKLA	32	PRIGC	35	SQEA	26
MAKLB	32	PRUG	37	SQEB	26
MAKX	33	PRUGCA	35	SQEC	26
MAKXA	33	RDR	28	SQED	26
MAKXB	33	READ	28	SQEF	26
MAKXC	33	READA	28	SQEH	26
MDICALL	24	READF	28	SQEK	26
MEM	22	RECALL	22	SRB	27
MOV	6	RES	24	SRT	27
MOVE	7	REW	11	SRTA	27
MOVK	7	RTRN	23	SRTF	27
MOVL	7	RTRNG	23	SRTG	27
MOVZ	6	RTRNK	23	START	4
MSGA	4	RY	8	STB	4
MSGB	24	RYC	8	TRA	9
PCBUF	29	RYD	8	TRAB	10
PCBUFA	29	SKP	12	TRAE	9, 29
PCBUFB	29	SKPB	12	TRAH	9
PRAG	36	SKPC	12	TRAK	10
PRAGA	36	SKPCA	12	TRAL	10
PRAGB	36	SKPE	12	TRALA	10

Index to Subroutines in EDITLIB Flowcharts

NAME	PAGE	NAME	PAGE
CKNAME	34	MSGB	24
COPY	29	PCBUF	29
DOCTS	22	PRIG	35
EXC	29	PRUG	37
GETD	23	RDR	28
GROW	38	READ	28
GROWA	38	RECALL	22
IOREAD	IO - P. 1	RES	24
IOWRITE	IO - P. 2	RTRN	23
LIST	24	SQA	25
LOC	23	SQB	25
MAKE	30	SQE	26
MDICALL	24	SRB	27
MEM	22	SRT	27

14.13 EDITSYM

EDITSYM first return jumps to ZERO to clear the dictionary and edit tables. Upon returning from ZERO, a return jump to SETFET is issued to read the input parameters and set up the file environment table (FET). After returning from SETFET, the list line counter (LINEL) and the output line counter (LINEO) are set to 55. NC is set to the address of the NEWCOM table. A check is made to see if an old program library exist. If an old program library does exist, return jump to COPYDC, if not, jump to ENTRY1.

ENTRY1

This section reads the correction input by use of the READIN macro. The flag that indicates a *DECK or *COPY control card is cleared and the primary sequence number (PSN) and the secondary sequence number (SSN) are set to display code zero. A return jump to BRKDWN is issued. If an EOR is encountered by READIN control goes to PROC; if not, return jump to COMPARE. If control card is a card other than *EDIT card, control goes to ENTRY20. If card is an *EDIT card, replace EDIT with COPY and return jump to WRITES, a check is made for illegal control card.

ENTRY20

If an error occurred, write control card and diagnostic on OUTPUT and go to ENTRY1. If the card is recognized as an EDITSYM control card, return jump to WRITES. Next determine which control card is read by means of CMPAR.

CMPAR equal to:

A. 0 designates *COMDECK control card; control goes to ENTRY7.

B. 1 designates *DECK control card; control goes to ENTRY4.

C. 2 designates *COPY control card; control goes to ENTRY4.

D. 4 designates *EDIT control card; control goes to ENTRY13.

E. 3 or 5-9 designates is a primary or secondary edit card, jump to ENTRY8.

ENTRY4

If this is the first time a text deck is edited, jump to PROC1A to write the common section on the new program library. If EDITSYM control card is a *COPY card, return jump to PROC3 and COPYA. Upon returning from COPYA, control goes to ENTRY19. If control card is *DECK return jump to DECKA. Upon returning from DECKA control goes to ENTRY19.

ENTRY7

This point is entered only if a *COMDECK is being processed. A return jump to COMDECK is issued and control goes to ENTRY19.

ENTRY8

A check is to see if CMPAR equals 3, if so, return jump to WEOR1. Upon returning from WEOR1 control goes to ENTRY19. If CMPAR equals 5 through 9 set to A2 to address of first primary sequence number (n2) and return jump to CONVERT. Set A2 to address of first secondary sequence number and

return jump to CONVERT. Store first converted sequence number in CONTROL array. Set A2 to address of second primary sequence number and return jump to CONVERT. Set A2 to address of second secondary sequence number and return to CONVERT. Store second converted sequence number in CONTROL array. Store editing status in CODE. Store code and beginning CARD address in third word of CONTROL entry.

ENTRY10

This section reads correction input by means of the READIN macro. If an EOR is encountered, write a diagnostic on output and return jump to ENDRUN. If card read is not control card, return jump to PACK and store packed card image in CARD array. Jump back to ENTRY10 to read next card. If card is control card, store last CARD address to complete third word of control entry, and jump to ENTRY20.

ENTRY13

This area is entered only when an *EDIT card is read. Set up input for EDTSORT and return jump to EDTSORT. Check for old program library, if it does not exist, write diagnostic on OUTPUT. If old program library exists return jump to SCANDIC. Store starting and ending address+1 of CONTROL entry in NEWCOM table, store control counter in ENDC. If deck name was not found in the dictionary, jump to PROCS. If deck name was found in dictionary, but text has been edited previously, write diagnostic on output and return jump to ENDRUN. If deck name was found in dictionary, and no text editing has been done, jump to ENTRY1 to read the next correction set.

ENTRY15

This section is entered if a *COMPILE card is read from correction input. If a compile file exists, write an EOR and rewind the file. In either case, put the new file name in the first word of the COMPILE FET. The control returns to ENTRY1 to read the next correction set.

ENTRY19

This area is entered as the normal return to ENTRY1 after an *EDIT for a text check is encountered. If a *COMPILE control card was read in the previous correction set, write EOR, rewind the file, and zero out first word of the COMPILE FET. In either case, control returns to ENTRY1.

PROC-PROC2

Control comes to this section if the deck to be edited is found to be a text deck. A check is made for entries to the NEWCOM table. If entries exist the NEWCOM address pointer (NC) is decreased by one. PROC is entered if an EOR is read from corrections input. If no text deck has been previously edited, the NEWCOM table is sort (return jump to EDTSORT). Upon returning from EDTSORT, a return jump to EDX is issued to write the corrected common section of the new program library. Then, return jump to COPYDC to copy the edited common section into central memory. At PROC2, control goes to GOOF, an entry point in ENDRUN, if an EOR was read from correction input, control goes to ENTRY5 if PROCS was entered by means of a *DECK or COPY control card.

PROC3

This routine is entered via a return jump and normally exits through entry point. PROC3 searches the OPL for decks specified on the *COPY control card. At PROC4 OPL is read by means of the READ macro. Records are read from OPL until the deck name at N2 matches an OPL deck names. If an EOF is read for the first time, the tape is repositioned at the first text deck. The second time an EOF is read, a diagnostic is written on OUTPUT, and a return jump to ENDRUN is issued to terminate the EDITSYM run.

ENDRUN

This routine is entered via a return jump or through entry point GOOF. An EOR is written on all files and an EOF is written on all files except OUTPUT by issuing a return jump to WEORF with A1 set to the address of the file FET. If the list file name is L or LIST, rewind the list file and copy to OUTPUT. Rewind COMPILE and exit via the ENDRUN macro.

WEORF

This routine is entered via a return jump and exits via the entry point. Input is A1 set to address of the first word of the FET. If the first word of the FET is zero, return; if not, write EOR on the file. If file is output, return; if not, write EOF.

SCANDIC

SCANDIC is entered via a return jump and exits through the entry point. Input to SCANDIC is the DICT array, N2, and B7 equal to one. If the first word of the dictionary is zero with N2. If N2 matches the dictionary entry, set B6 equal to non zero, and A1 equal to address of common deck in common array. If dictionary is exhausted and N2 is not a DICT entry, set B6 equal to zero.

EDTSORT

Entry to this routine is made via a return jump and exit is always through the entry. Input to EDTSORT is A0 equal address of table to be sorted, B4 equal number of words in the table plus one, X3 beginning word in table to be sorted, and B1 equal to number of words per table entry. Sorting requires two words of intermediate storage, EDSTMP and EDSTMP+1. Two values of the table are compared, and the smaller entry placed in the higher table position. Succeeding values are compared with values above them in the list (i.e. sorted values) starting with the last sorted entry. If the trial entry is smaller, the larger entry is moved down. If the trial entry is larger or equal to the sorted entry, it is stored below.

WEOR1

WEOR1 writes an EOR on the COMPILE file. It is entered via a return jump. If the first word of the COMPILE FET is zero, write a diagnostic on OUTPUT and return jump to ENDRUN, if not, write EOR on the COMPILE file and return.

BRKDWN

This routine is entered via a return jump and always exits via the entry. Input is EDTCRD through EDTCRDZ and X1 from the READIN macro. At BRKDWN, flags and storage areas are initialized to zero and A6 is set to store the first parameter. B6 is set to number of characters in card image if an EOR was read by READIN return with B1 equal to 2. BREAK1 isolates each character. If character is not a separator, the next character is positioned and control goes to BREAK1. If character is right parenthesis, go to BREAK4, left parenthesis go to BREAK5, decimal go to BREAK9, comma go to BREAK5, blank go to BREAK3 or equal sign go to BREAK12. BREAKS reads the next word and checks for ended working area. If control goes to BREAK4, all parameters except the last have been stored, X0 is set to five. Control goes to BREAK5. If this is the first time through or the decimal flag is set, go to BREAK6. If not set B3 to one. At BREAK6 if parameter to be stored is zero go to BREAK7.

If not, left justify parameter. At BREAK7 store parameter at N(i) and increment X0 by one. If B3 is set to zero or BRKDWN was called from SETFET, go to BREAK8; check if all parameters have been stored. If they have, control goes to BREAK11. Otherwise, clear the storage area and go back to BREAK3. BREAK9 sets the decimal flag and goes to BREAK5. BREAK11 is the normal exit from BRKDWN. N2 is stored in UNPKNAM and BRKDWN8 returns. BREAK12 is reference only if processing the EDITSYM call card. The parameter preceding the equals sign is stored with bit 0 set. If the equal sign is the last character in a word, control goes to BREAK5; if not, control goes back to BREAK1.

COMPARE

This routine is entered via a return jump and always is exited through the entry. The contents of N1 is determined and B5 is set to indicate the result as follows:

B5	Contents of N1
0	*COMDECK
1	*DECK
2	*COPY
3	*WEOR
4	*EDIT
5	*INSERT
6	*DELETE
7	*ADD
8	*CANCEL
9	*RESTORE
10	*CATALOG
11	*COMPILE
12	error

At COMPAR1, CMPAR is set to B5 and the routine returns.

CONVERT

The function of this routine is to convert display code decimal numbers to binary. It is entered via a return jump and is returned either through the entry or ENDRUN. CONVERT initializes the registers, then CNVRT1 isolates the character. Finding a zero sends control out the entry; finding a special

character sends control to CNVERT2 which return jumps to EPLIST and exits to ENDRUN.Finding an alphanumeric character sends control to CNVERT3 which left justifies the input regiser X2 and exits. All numbers are converted and added to the X0 register and the first zero terminates the number.

PACK

This routine packs one card image by removing blank characters and inserting 55nn in their place where nn is the number of blanks minus one. Input to PACK is as follows:

B1 = address of 8 word block containing the unpacked card image, the first character of which is in bits 54-59 of the first word. The assumption is made that the card has been read using READIN so that 8 words are occupied, billed out with blanks.

B2 = address of 13 words to be used for the packed card image

B4 = editing status

B7 = 1

Output from PACK is as follows: The card is pack and placed in the area specified. B2 = address of word containing the last character of the packed card image. B1 = LWA+1 of unpacked card image.

This routine is entered via a return jump and always exits through the entry point. PACK05 sets B6 to number of characters in a word at PACK10 isolate a character. If character is a blank, control goes to PACK30; if pack count is non-zero, control goes to PACK50. PACK15 stores the character in X5 and checks if X5 is full. If X5 is full, control goes to PACK20; if not, control goes to PACK25. PACK20 stores X5 in B2 and increments B2. PACK25 decrements B6 by one. If input word is exhausted control goes to PACK45; if not, the input word X1 is positioned for next character and control goes to PACK10. At PACK30, if the blank is stored in X5 and if X5 is full, go to PACK40; if not, position X5 for next character. PACK35 increment the blank counter by one, and control goes to PACK25. PACK40 stores X5 in B2 and increments B2 by one. Control goes to PACK35. PACK45 increments B1 and checks if end of input area. If not, control goes to PACK05. If blank count is zero control goes to PACK70. At PACK50 if blank count is less than 77B control goes to PACK55. Otherwise 77B is stored as blank count, and a return jump to PACK80 is issued. The remaining blank stored and a 55B is stored in X5. A return jump to PACK80 is issued. At PACK55 the blank count is stored in X5. If X5 is full control goes to PACK60, if not, position X5 for next character and jump to PACK65.

PACK60 stores X5 in B2 and increments B2 by one. PACK65 checks if scan is completed. If it is, control goes to PACK70; if not, control goes back to PACK15. At PACK70 jump to PACK75; if X5 cannot hold editing status. If there is room for editing status, store status in B2 and return. PACK75 increment output word pointer and stores edit status and return.

PACK80

PACK80 is entered via a return jump and exits through the entry. This routine is called by PACK to make room for character in X5. If X5 is full, control goes to PACK85, if not X5 is shifted left one character and is returned to PACK. PACK85 shifts X5 one character to the left and stores X5 at B2. B2 is incremented and PACK80 exits.

SUMPR

SUMPR is called from SUM to write a message on OUTPUT. Input is A1 set to address of working storage. X1 is stored in OUTPUT+5 and a return jump to IOWRITE is issued. SUMPR exits through its entry.

SUM

This section is called when a *CATALOG control card is read. First, a return jump to SUMPR is issued to eject a page. Store first word of EPL FET in SUMH and files name in COPYDC1. Return jump to COPYDC3 to position at beginning of file specified by N2 and read file (return jump to COPYDC2). If IN is not equal to OUT control goes to SUM6. If not, check for EOR or EOF. If an EOR or EOF is read, control goes to SUM8. Otherwise, a return jump to ENDRUN is issued and the run is terminated. At SUM6 the first word of a record is neither a common or a text deck identification word. A diagnostic is written on OUTPUT and control goes to ENTRY1. The deck name and edition number are extracted from the ID and a return jump to SUMPR is issued. This process continues for each record until an EOF is encountered. If the decks processed are common decks, the process continues until another EOF is read.

UNPACK

This routine is called via a return jump. Input is as follows:

B1 = address of the packed card image

B2 = address of a nine word block to be used for output from the routine (i.e. for the unpacked card image)

B7 = 1

A0 = last word address + 1 of the pack card image area. The maximum address to be used in searching for the edit status (i.e. the end of the pack card image is either

UNPKERR must be initially set to zero
PSN and SSN must initially be set to the constant 00000000003333333333B

Output from UNPACK is as follows: If UNPKERR=0

A6 = address of last word stored

B1 = last word address of the packed card image

The unpacked card image is stored in the location specified by B2 as follows:

Word	Contents
1-7	Columns 1-70 of the card image
8	Columns 71,72 of the card image, 7 character deck name, first digit of primary sequence number.
9	The remaining 4 characters of the primary sequence number, followed by a decimal point and the secondary sequence number, if any, if there is no secondary sequence number, display code blanks are inserted.

If UNPKERR is now zero, either A0 or B1+12 is reached before edit status is found.

The packed card image is moved from the input address defined by B1 to a temporary block the pack subroutine. If the edit status is not found, the flag UNPKERR is set to n where n is the number of words moved to the temporary block in searching for the edit status. If n is greater than or equal to 13, there is an error in the packed card image; if n is less than 13, the packed card image is incomplete. If the edit status is found, UNPKERR is set to zero and the card image is unpacked. Each 55nn encountered is converted to nn+1 blanks. A4 is set to UNPRNAM and a return jump to UNPACK80 is issued when a 00 character is found. If unpacked card is being added or inserted and a NPL is not requested, the asterics are inserted in place of the secondary sequence number, otherwise, blanks are stored in the secondary sequence if it does not exist. In either case, the sequence numbers are stored and UNPACK returns.

UNPK80

This routine is entered via a return and always exits through entry. Input is X4 set to the deck name and A6 equal to address of last word stored. UNPK80 sets up the column 71 and 72 of the card image, the deckname, PSN, and SSN. If the card is being added or inserted and a NPL is not requested, asterisks are inserted for PSN. Store seventh word, set X6 to eighth word and return.

COPYDC2

COPYDC2 reads from the file specified by COPYDC1. It is entered by a return jump and exits through the entry. The FET address is obtained from the contents of COPYDC1. IN and OUT are set equal to FIRST, a return jump to CPC is issued to read. Upon returning from CPC, B6 is set equal to IN, B5 to OUT, and OWRK to the first of the buffer; then, COPYDC2 return.

COPYDC3

COPYDC3 positions the file. It is entered via a return jump. The file address is obtained from the contents of COPYDC1. A return jump to CPC is issued to skip records backwards until a level 17 EOR is encountered. If DICT is zero, a return jump to CPC issued to skip one record forward. If DICT is non-zero, a return jump is issued to skip backwards the number of records in the COMMON section.

COPYDC

The purpose of this routine is to copy the common section to central memory. COPYDC is called via a return jump with an input word following the return jump containing the address of the FET of the file to be copied. The address is stored in COPYDC1 and the return address incremented by one. At COPYDC7 a return jump to COPYDC2 is issued. If IN does not equal OUT, control goes to COPYDC6. If an EOR or EOF was read, control goes to COPYDC8. Otherwise a diagnostic is written and a return jump to ENDRUN is issued to terminate the run. COPYDC6 makes a check to see if the deck read is a COMMON deck; if so, control goes to COPYDC4. If DICT is non zero, a diagnostic is written and a return jump to ENDRUN is issued to terminate the run. If DICT is zero, a skip record backward is issued to position the file specified at the beginning of the text section. At COPYDC4, the dictionary entry for the common deck is created and the common deck counter in DICT is incremented. COPYDC5 copies the common deck into COMMON and a return jump to COPYDC2 is issued. This process continues until an EOR is read. When an EOR or EOF is read, a check is made to see if this is the end of the common section. If it is, DICT is completed and the routine returns. If not, control goes back to COPYDC7.

DECKA

This routine is entered via a return jump and normally exits through the entry point. DECKA is called to add text decks to NPL. Input is N2 containing deck name; N4 containing level number for deck. If the level is less than two, it is set to two. The level number is stored as the last 12 COMMON of the first ID prefix word. Next, the second ID prefix word is created containing the deck name and the edition number and the two prefix words are written on NPL. Subsequent cards are read from the correction input until an *END is read. For each card read, a return to GRAPH, GRAM, and EJECT1 is issued to write the compile and list files. A return jump to SHOW is issued to write NPL. If the card is a *CALL card, a return jump to CALL is issued. The read cycle continues until an *END card is read. Then, a return jump to SHOW is issued to write the *END card on NPL and the routine returns. If an EOR or EOF appears before the *END card, a diagnostic is written by means of ERLIST and a return jump to ENDRUN is issued to terminate the EDITSYM run.

CALL

This routine is entered via a return jump and normally exists through the entry point. Its purpose is write a common deck on the compile file when *CALL is read. Return jumps to BRKDWN and SCANDIC are issued. If the deck name on the *CALL card is not in the dictionary write a diagnostic and return. Otherwise, the sequence numbers are set to zero and the text sequence numbers saved. Card images are read from COMMON unpacked by UNPACK and written on the compile file until an *END card is read. GRAM is called to do the actual writing of the compile file. When an *END card is encountered, the sequence numbers are restored and the routine returns.

ERLIST

This routine is entered via a return jump and always ecits through the entry point. The word following a return jump to ERLIST contains the start of the diagnostic to be printed. If the word following the diagnostic is nonzero, the card image in EDTCRD through EDTCRD+9 is printed preceding the diagnostic. If the word following the diagnostic is zero, only the diagnostic is printed.

COMDEK

This routine is entered via a return jump and normally exits through the entry point. COMDEK creates prefix words 1 and 2 in the common section and updates the prefix word count in the dictionary. Then, a dictionary entry is stored containing the common deck name and a pointer to prefix word 1 in the common section and the common deck is read from input to the common section. If an error occurs a diagnostic is written and a return jump to ENDRUN is issued to terminate the run. COMDEK is called only if a *COMDECK is read from input.

COPYA

COPYA is called when a *COPY or a *EDIT preceded by no corrections is read from input. The routine is entered via and normally exits through the entry point. If an error occurs a diagnostic is written and a return jump to ENDRUN is issued to terminate the run. When COPYA is called, the old program library is positioned at the deck specified from OPL to NPL, COMPILE, and LIST files. *COPY, dn1, dn2 copies from dn1 up to and including dn2. *COPY,*, dn copies from the point OPL is positioned up to and including dn. *COPY, dn,* copies from dn to an EOF. *COPY,*,* copies from the point OPL is positioned to an EOF.

GRAPH

GRAPH is entered by a return jump followed by a word containing the address of an unpacked image. The unpacked card image is moved to KATAB through KATAB+13. If the card image is not 14 words, the remaining words are blank filled. GRAPH always exits through the entry point.

WRITES

WRITES is entered via a return jump and always exits through the entry point. The routine writes a card image contained in EDTCRD through EDTCRD+8 on the OUTPUT file by use of GRAPH and GRAM.

GRAM

GRAM is entered by a return jump followed by a word containing the address of the FET for the file to be written and the location of the line image addresses. The line image is written by issuing a return jump to IOWRITE. GRAM always exits through the entry point.

BOOT

BOOT is entered via a return jump and always exits through the entry point. This routine sets up calls to GRAPH and GRAM to write the card image contained in EDTCRD through EDTCRD+9 on the output files. If the deck being written is a text deck, GRAM is called to write both LIST and COMPILE files.

If the deck being written is a common deck, GRAM is called to write only the LIST file. If a *CALL card is encountered EDTCRD through EDTCRD+8 is stored in STCARD and the deckname is saved. A return jump to CALL is issued. Upon returning from CALL, the deck name and EDTCRD are restored. If the card is cancelled, the card is written on the list file with CANCELLED in EDTCRD+9.

SHOE

SHOW is entered via a return jump and exits through the entry point. This routine calls PACK to pack the card image contained in EDTCRD through EDTCRD+7. Upon returning from PACK, the packed card image is written on the new program library by means of the WRITOUT macro.

GROW

GROW is entered via a return jump and exits through the entry point. If GROW2 is zero, the deck being processed is a common deck. If GROW2 is non zero, the deck is a text deck. In the case of a text deck, the old program library is read by use of the READ macro. If the deck is a common deck the central memory addresses are preset in GROW2+1 and GROW2+2, and the card image stored in EDTCRD. Contents of the EDTCRD array is unpack. If the complete card image of a text deck, is not unpacked another read is issued and the remainder of the card unpacked. If the end of the card is still not found or if the deck is a text deck, a diagnostic is written and a return jump to ENDRUN is issued to terminate the run.

STEP

STEP is entered via a return jump and exits through the entry point. The purpose of STEP is to increment a sequence number contained in INNUM. UNPKES is 2 for secondary sequence number and 1 for primary sequence number. For a secondary sequence number, a one is added to bit 0. For a primary sequence number, a one is added to bit 18. The incremented sequence number is stored in INNUM.

FEET

FEET is entered via a return jump. This routine checks EDTCRD for a *END card. If the card image in EDTCRD is not a *END card a return jump to BOOTY is issued. In either case, a return jump to SHOE is issued and PSN and SSN are saved in XPSN and XPSN+1.

EJECT1

EJECT1 is called before a line is written on the list file. The routine is entered via a return jump, and always exits through the entry point. A check is made to see if the list line counter (LINEL) is equal to 55. If it is not, LINEL is incremented by one and the routine returns. If LINEL is equal to 55, the page number is incremented by one and stored in the list heading. The page is jected and the list heading written. A return jump to BLANK is issued and LINEL is reset.

EJECT3

EJECT3 is called before a line is written on the OUTPUT file. The routine is entered via a return jump and always exits through the entry point. A check is made to see if the OUTPUT line counter (LINEO) is equal to 55. If it is, LINEO is reset and heading is written on the OUTPUT file followed by two blank lines. If LINEO is not equal to 55, LINEO is incremented by one and the routine returns.

BLANK

BLANK is called by a return jump from the eject routines and always exits through the entry. If BLANK is called from EJECT1, two blank lines are written on the list file. If BLANK is called from EJECT3, two blank lines are written on the OUTPUT file.

EDX

The main purpose of EDX is to set up input and calls to the editing routine EDY. Input to EDX is GROW2, NEWCOM, COMCNT and DICT. The contents of GROW2 is stored in WILCO and the address of NEWCOM is stored in MYA. If GROW2 is zero, (i.e. the COMMON section is to be edited), MYP is set to the number of common decks and MYO is set to the address of the first dictionary entry. The NEWCOM entry is broken down such that MYC contains the address of the next correction trio, MYB contains the last word address+1 of the correction trios for that specific deck, and MYALPHA contains the pointer to the common deck to correct. If the deck to be edited is a text deck, MYB and MYC are set but MYALPHA is not. NEWED is set to one and a return jump to EDY is issued. Upon returning from EDY if GROW2 is non zero, the routine returns. Otherwise, MYA and MYO are incremented and MYP is decremented by one. If MYP is zero and all the NEWCOM entries have been accounted for the routine returns. If MYP is non zero and all the NEWCOM entries have been accounted for, EDY is called to write the remaining common decks on NPL. Otherwise, MYA, MYB, and MYC are reset the the process begins again.

EDY

EDY edits COMMON and TEXT decks and sets up calls to FEET and GROW to write out NPL, COMILE and LIST files. If the deck to be edited is a text deck, MYALPHA is set to OPL, OUT. IN is stored in GROW2+2 and IN+2 is stored in GROW2+1. In case of a COMMON deck, set GROW2+2 to zero and set GROW2+1 to the address of the second prefix word. The deck name is stored in UNPKNAM and PSS and SSN are set to zero. The edition number is incremented if needed and written on NPL. If no corrections are to be made, return jump to GROW and FEET are issued until a *END card is found by GROW. A final return jump to FEET is issued to write the *END card on NPL. A blank line is written on the LIST file. An EOR is written on NPL and the routine returns. If corrections are to be made, FLAP is set to the first CONTROL entry, FLG to the second CONTROL entry, and FLACK to the third. A return jump to GROW is issued. If an *END is encountered by GROW a diagnostic is written, an EOR is written on NPL and the routine returns. Otherwise, the sequence number is incremented and compared to FLAP. If FLAP is larger than the sequence number, the card is written out and the process continues.

If FLAP is larger than the sequence number, a diagnostic is written, the card written out, and the next correction located. If FLAP and the sequence number are equal, FLACK is broken down such that MYN contains the last word address of the packed inserts, MYM contains the first word address of the packed inserts. The code is checked, and the section of the routine is jump to that corresponds to the code.

The code equal to 0 indicates an insertion and the following path is followed. A return jump to FEET is issued. Upon returning from FEET, GROW2 is set to zero and MYJ is set to the original contents of GROW2. Also the contents of GROW2+1 and MYM are exchanged and the contents of GROW2+1 and MYN are exchanged. Return jumps to GROW and FEET are issued until all the packed cards have been inserted. MYJ, MYM, MYN, GROW2, GROW2+1, and GROW2+2 are restored to their original value. The routine goes back to search for the next correction for this deck.

The code equal to 1 indicates a deletion and the following path is taken: A deletion message is written on the list file and then the cancellation path is taken.

The code equal to 2 indicates an addition and the same path follows as for insertion.

The code equal to 3 indicates a cancellation and the following path taken: The editing status is stored in MYJ. If the editing status is 0, the card is delted. If the edit status of the card is 2, the card is deleted. If the edit status of the card is something other than 2, the card is cancelled (i.e. written with edit status 1) if a second sequence number does not exist (i.e. FLAG = 0) or if the second sequence number and the card sequence number are equal, PSN and SSN are restored and the routine goes back to pick up the second correction. If the sequence numbers are not equal, a return jump to GROW is issued. If an *END is found by GROW a diagnostic is written and the next correction is picked up. If not, the card sequence number is incremented and the search for the end of the cancellations or deletions continues.

The code equal to 4 indicates that cards are to be restored. All cards that have been cancelled in previous EDITSYM runs are restored (i.e. all cards with edit status of 2 are permanently deleted).

EDYX

EDYX is entered via a return jump followed by a word containing the address of the diagnostic and exits through the entry point. The purpose of EDYX is to write diagnostics on the list and OUTPUT files by setting up calls to GRAPH and GRAM. The code is extracted from FLACK and used to obtain the correct name and shift count from the EDYXA table. The primary and secondary number in FLAP and the primary and secondary numbers in FLAG are converted to display code. A diagnostic is written on the list and OUTPUT files consisting of the control card followed by the error message.

EDYW

EDYW is entered via a return jump and exits through the entry point. This routine is called from EDYZ to shift the contents of X7 and store it in EDYXB.

EDYZ

EDYZ is entered via a return jump and exits through the entry. This routine is called from EDY convert the binary sequence numbers in FLAP and FLAG to display code.

BLNKLIN

This routine is entered via a return jump and exits through the entry point. KATAB through KATAB+14 is filled with blanks and the blank line image is written on the list file.

ADPCT

ADPCT is entered via a return jump and exits through the entry point. This routine increments a display coded number by one. Input is A0 equal to the address of the number to be incremented. Output is A0 equal to the address of the incremented number.

ZERO

ZERO is called from EDITSYM to initialize tables and buffer areas to zero. It is entered by a return jump and exits through the entry.

SETFET

SETFET is called from EDITSYM to set up file names in the FET. The EDITSYM call card is broken down by the BRKDWN routine thus storing each parameter in N(i). Each N(i) is compared with a TABLE1 entry until the entries match or the N's are exhausted. If N(i) and a TABLE1 entry match and bit 0 is not set, N(i) is stored in TABLE2. If N(i) and a TABLE1 entry match and bit 0 is set, N(i+1) is stored in TABLE2. The next TABLE1 entry is picked up and the search continues. When TABLE1 entries are exhausted, TABLE2 entries are stored in the corresponding FET's.

ROUTINE	ROUTINES REFERENCED
ADPCT	
BLANK	GRAM
BLNKLIN	EJECT1, GRAM
BOOT	GRAPH, GRAM, EJECT1, CALL
BRKDWN	
CALL	BRKDWN, SCANDIC, UNPACK, GRAM, ERLIST, ENDRUN
COMDECK	ERLIST, ENDRUN, PACK
COMPARE	
CONVERT	ERLIST, ENDRUN
COPYA	COPEXC, UNPACK, BOOT, BLNKLIN, ERLIST, ENDRUN
COPEXC	
COPYDC	COPYDC3, COPYDC2, ERLIST, ENDRUN

Routine	Routines Referenced
COPYDC2	
COPYDC3	
DECKA	UNPK80, ADPCT, GRAPH, GRAM, SHOW, CALL, ERLIST, ENDRUN
EDTSORT	
EDX	EDY, ENDRUN
EDY	GROW, FEET, BLNKLIN, STEP, EDYX, ENDRUN, EDRESB, EJECT1
EDRESB	EDYX
EDYX	EDYZ, GRAPH, EJECT1, GRAM, EJECT3
EDYW	
EDYZ	EDYW
EJECT1	ADPCT, GRAM, BLANK
EJECT3	GRAM, BLANK
ENDRUN	WEOF
ERLIST	GRAPH, GRAM
FEET	BOOT, SHOW,
GRAPH	
GRAM	
GROW	ERLIST, ENDRUN, UPACK
PACK	PACK80
PACK80	
PROC3	ERLIST, ENDRUN
SCANDIC	
SETFET	BRKDWN, ERLIST, ENDRUN
SHOE	PACK
STEP	
SYMPR	
UNPACK	ADPCT, UNPK80
UNPK80	
WEOF	
WEOR1	
WRITES	GRAPH, GRAM, EJECT3
ZERO	
EDITSYM	ZERO, SETFET, COPYDC, BRKDWN, COMPARE, WRITES, ERLIST, ENDRUN PROC3, COPYA, DECKA, COMDECK, WEOR1, CONVERT, PACK, EDTSORT, SCANDIC, EDX, SUMPR, COPYDC3, COPYDC2.

14.14 IO

General

IO is a CP program, occupying about 160B words. It is called by any CP program that has IOREAD, IOWRITE, IOIO or IOSAV as an external. The macros

```
READIN efm
```

and

```
WRITOUT efm
```

call IO indirectly through IOREAD and IOWRITE. The macros

```
READIN lfn,/name/
READIN lfn, number
WRITEOUT lfn,/name/
WRITOUT lfn, number
```

dealing with random files, call program IORANDM, which in turn calls IO.

Function

IO carries out the blocking and deblocking, and for BCD files the suppression and insertion of trailing blanks, needed for "reading" or "writing" a delimited area of central memory, known as the "workspace", which is separate from any circular buffer.

Entry Information

The calling sequence for non-random reading

```
RJ IOREAD
VFD 60/lfn
```

For non-random writing it is

```
RJ IOWRITE
VFD 60/lfn
```

where lfn is the address of the first word of the FET. The FET must be at least 6 words long, and its 6th word must be

```
VFD 30/a,30/b
```

where a is the address of the first word of the workspace, and b-1 is the address of its last word. For random reading and writing, see the notes to program IORANDM.

Exit Information

On return from executing a non-random macro, X1 contains 0 unless a read was called for and could not be fully carried out. If the buffer was empty and reading the file found an e.o.f., X1 will be negative. If an e.o.r.

was found before the workspace could be filled, X1 will contain the address of the first unfilled address of the workspace. For a BCD file, this must be the address of the first word of the workspace, and the workspace will be full of blanks, because if even one word at the end of the record was transferred to the workspace, the rest of the workspace is then filled with blanks and a normal read is considered to have taken place.

Other Programs Called

CPC.

Registers Destroyed

A1, X1, A6, X6. Note that A7 and X7 are not touched.

Narrative

The main routines are IOREAD and IOWRITE, which are entered like subroutines. But both construct their final exit at IORE, and exit through it, not through IOREAD or IOWRITE.

IOREAD begins by calling subroutine IOSAV with 1 in the calling sequence, indicating that the calling sequence to IOREAD contains 2 words. IOSAV saves the registers except A1, X1, A6, X6, A7, X7, sets up the final exit at IORE, sets B3 to the first word of the workspace and B2 to its last+1 word. If the FET is less than 6 words long, or if the workspace seems to have a negative length, we abort with a message at IOSVA. If the workspace seems to have a zero length, we really do nothing, and do the final exit with X1=0. On exit from IOSAV back to IOREAD we set X0=10B to indicate reading, and go to IOIO.

IOWRITE begins by calling subroutine IOSAV just as for IOREAD, but then puts 14B in X0 to indicate writing and goes to IOIO.

At IOIO we set B6 to X0-10B, i.e. zero for reading and non-zero for writing. Then extract the mode bit from the first word of the FET, and set B7 to zero for BCD, or non-zero for binary. Then set B4 to minus the FIRST pointer in the FET, and B5 to minus the LIMIT pointer. Now we have set all B-registers to values they will keep throughout, except: B2- if writing BCD, B2 will be adjusted to point to the word next after the last non-blank word in the workspace; B3- when emptying or filling the workspace, B3 will be stepped from left to right until it equals B2.

Now if writing, we go to IOIOA, whence, for binary, we go straight to IOO. For BCD, we reduce B2 if necessary until it points to the word after the last non-blank word in the workspace, or to the second word if the whole workspace is blank.

If reading, then for binary we go straight to IOI. For BCD we fill the workspace with blanks (routine around IOIOB) and then go to IOI.

At IOO, we begin moving data from the workspace to the buffer. If B2 = B3, we are finished moving and go to IOOF. Otherwise, put the next word from the workspace in X3, and step B3. Check that the buffer is not full, and go to IOOD. If full, call subroutine IOISS with X4 = 22B, to write out the buffer with recall. Then return to IOOB to pick up the FET pointers again.

At IOOD, store the word from the workspace in the buffer, and step the FET IN pointer. Then return to IOO to repeat the cycle until the workspace is emptied; unless BCD, in which case do not return to IOO if the last byte of the word was zero, This ends the workspace even if there is a non-blank word to the right of it; and we go straight to IOIK. When we come to IOOF, because B2 = B3, we can go straight to IOIK if binary. But if BCD, we must force the line to end with a zero byte. If the last word, which is still in X3, ends in two blanks, go to IOOFA. Otherwise, put the word 5555B in X3 and go back to IOOB. The word will get put into the buffer, and as it ends in two blanks, we will get to IOOFA. At IOOFA, we modify the last word stored by changing its last two characters to a zero byte, and store it again where we stored it before. If it was the extra word 5555B, it will become a whole word of zero. Then go to IOIK.

In reading, we begin at IOI to move data from the buffer to the workspace. If the buffer is not empty, go to IOIC, where we put the next available word in X3, and step the FET OUT pointer. At IOID, we normally find that B6 = 0, because we are reading; so do not go to IOIF. Then set X2 = 0 if the last word ended in a zero byte, put the last word unchanged in X6, and set X5 = 1 to show at least one word has been read (it was zeroed just after IOIO if BCD mode, and is not otherwise altered). Now if binary, go to IOIE to store the word in the workspace, step B3, and go back to IOI if the workspace is not yet full, or to IOIK if it is full. If BCD, and X2 = 0, we have the last word of the line, and we go to IOIG. Otherwise proceed as above to IOIE. At IOIG, we replace trailing zero characters in the word in X6 by blanks, then store it in the workspace, and set B3 = B2 to show the workspace is effectively full. Any extra words in it were blanked above at IOIOB. Then go to IOIK.

At IOIE, we stored a word that did not end in a zero byte in the workspace, stepped B3, and returned to IOI if the workspace was not full. If it was full, and the mode is binary, simpy to to IOIK. But if BCD, we must put no more in the workspace, but skip through the buffer until we have seen a word that ends in a zero byte. The words so skipped will never get into the workspace. We set B6=1, instead of 0, its normal value for reading, and return to IOI. There we get the next word of the input record, step the FET OUT pointer, and reach IOID. Now because B6=1, go to IOIF. Here we check the word in X3 for final zero byte; if yes, we have skipped enough; restore B6=0 and go to IOIK. If no, go back to IOI to skip another input word.

Before describing IOIK, we must return to IOI and the case of an empty buffer. If the FET status is end-of-information, or end-of-file, go to IOIMA, set B3 = -1, and go to IOIY. Otherwise, if end-of-record, go to IOIY with the workspace pointer still in B3. But if BCD, and if X5 is not zero, showing that at least one word has been moved to the workspace, go

on to IOIK as if the workspace had been filled. This is for the case that the last word of a record does not end in a zero byte, though normally it should. At IOIY, if B3 = B2 we have filled the workspace and can go off to terminate at IOZZ without regarding the end-of-record until the next time IOREAD is called. This could only happen if we had a zero-length workspace, empty buffer, and e.o.r. status. Then the call on IOREAD would achieve exactly nothing. If B3 is not = B2, we store B3, which is -1 for e.o.f. or the workspace pointer if e.o.r., at IOZW, to be put in X1 on exit. Then zero bit 4 of the first word of the FET unless the FET shows a random file, and go to IOZZ. We zero this bit so that by repeated READIN calls, without touching the FET, the user can read across record and even file boundaries. The user will be notified of them by the content of X1 on return, but IOREAD will handle the transitions. However, for a random fi le, there is no merit in going to the next record in chain sequence, so we do not zero the bit.

If the buffer is empty but the FET status is not e.o.i., e.o.f., or e.o.r., we go to IOIA, to call subroutine IOISS with X4 = 22B. This reads with recall. Then arrive at IOI to attack the buffer again.

We come to IOIK when we have completed the read or write request. If possible we would like to issue a request for CIO to read or write without recall, to keep the file moving between now and the next time IOREAD or IOWRITE is called. But if the FET status is busy, there would be no point to it, so go straight to IOZZ. Otherwise, if writing, we see whether the buffer is at least half full. If so call subroutine IOISS with X4 = 20B (write without recall), then go to IOZZ. If reading, and the FET status is not e.o.r. or e.o.f., we go to IOZZ if the buffer is not at least half empty. If the buffer is at least half empty, call IOISS with X4 = 20B (read without recall) and then go to IOZZ. But if the FET status is e.o.r. or e.o.f., then if X5 is not = 0, we have put something in the workspace and go to IOZZ. The next call on IOREAD will return the e.o.f. or e.o.r. status to the user. But if X5 = 0, we have put nothing in the workspace, and go to IOIM to set up the e.o.f. or e.o.r. response. It does not seem possible that the branch to IOIM could take plase here.

At IOZZ we are ready to exit from IOREAD or IOWRITE. First call subroutine CPC04 to restore all registers except A1, A6, A7, X1, X6, X7. They were stored at IOREG through IOREG+8 when we initially called subroutine IOSAV. Then put IOZW (0 for normal, -1 for e.o.f., workspace pointer for e.o.r.) in X1, zero IOZW for next time, and take the exit at IORE, which was set up by subroutine IOSAV.

Subroutines

We do not count IOREAD and IOWRITE as such, because they form the main part of IO, and are not exited normally.

IOSAV

This is called on entry to IOREAD and IOWRITE with 1 in the calling sequence, showing that the calling sequence to them is 2 words; and on entry to IORR and IORW (in program IORANDM) with 2 in the calling sequence, showing that the calling sequence to them is 3 words.

First we call subroutine CPC03 (in program CPC) to save all registers except A1, A6, A7, X1, X6, X7 in IOREG through IOREG+8, and set B1=1. Then find the word from which IOSAV was called, and assume that the word before is the ordinary exit word of the subroutine (IOREAD, IOWRITE, IORR, or IORW) that called IOSAV. Add the number in the calling sequence to IOSAV, plus 1, to that outer return address, and store a jump to that result in IORE. So IOREAD, IOWRITE, IORR, and IORW all exit through IORE.

Then check that the FET to which the calling sequence for IOREAD, IOWRITE, IORR or IORW points has at least 6 words, and that the 6th word does not suggest a negative length workspace. Abort through IOSAV otherwise. Then set up registers as below in "Exit Information".

Entry Information

Only the calling sequence, as follows:

```
RJ IOREAD or IOWRITE
VFD 60/lfn
```

where lfn is the address of the first word of the FET. Then, at IOREAD or IOWRITE:

```
DATA 0
RJ IOSAV
JP 1
```

or

```
RJ IORR or IORW
VFD 60/lfn
VFD 60/m or 60/0H name
```

where m is the number, or "name" the name of the record. Then, at IORR or IORW:

```
DATA 0
RJ IOSAV
JP 2
```

Exit Information

B1 = 1, A0 = address of FET first word, B3 = address of workspace first word, B2 = address of workspace last word + 1. X4 points to the first parameter in the call to the routine that called IOSAV (this is used only by subroutine IORB in program IORANDM).

Subroutines Called

Only CPC999, to abort.

IOISS

This is called with X4 = 22B for a read or write with recall or with X4 = 20B for a read or write without recall. X0 is assumed to contain 10B or 14B, for reading or writing respectively.

Entry Information

X0 = 10B for reading or 14B for writing. X4 = 22B for recall or 20B for no recall. The FET is assumed correct.

Exit Information

None.

Subroutines Called

CPC.

Registers Destroyed

A1, A6, X1, X4, X6.

14.15 IORANDM

General

IORANDM is a CP program, about 140B words long. It is called by macro:

```
READIN    lfn,m
READIN    lfn,/name/
WRITOUT   lfn,m
WRITOUT   lfn,/name/
```

which are assembled as:

```
RJ        IORR
VFD       60/lfn
VFD       60/m

RJ        IORR
VFD       60/lfn
VFD       60/0Hname

RJ        IORW
VFD       60/lfn
VFD       60/m

RJ        IORW
VFD       60/lfn
VFD       60/0Hname
```

respectively, where lfn is the address of the first word of the FET of a random file, m is the number of a record in the file, and "name" is the name of a record in the file.

Function

IORANDM, through its entry points IORR and IORW respectively, is used to begin reading or writing a record in a random file, and to move the first workspace-full. It behaves rather like IO, but manipulates an index, and puts into the 7th word of the FET, before actual reading or writing is begun, either the disk address of the record (for reading) or the address in which the disk address is to be stored (for writing). After the first workspace-full, one must use IOREAD or IOWRITE to continue reading or writing the same record. To terminate a record being written, a macro "WRITER" (calling CPC directly) should be used.

However, if IORW is called to write a record, and the status of the FET is write but not write end of record, the end of record will be written before anything else is done. Whenever a record is begun by IORW, the FET status is set to write completed even though no call for physical writing may have taken place, in order that the above safeguard may always work.

Entry Information

Only the calling sequence, as illustrated above under "General".

Exit Information

On return from executing the macro, X1 contains 0 unless a read was called for and could not be fully carried out. Then X1 will contain the address of the first unfilled word in the workspace. In the case of a BCD file, this will necessarily be the first word of the workspace. Note that an end-of-file return will never be given. Note also that if an end-of-record return is given, and IOREAD is then called to try to read the next record, it will merely return another end-of-record indication. For a non-random file, IOREAD would then cross the record boundary, but not for a random file. It would be necessary to do a random read to continue reading.

If the file is not random, or if there is no index or a zero-length index in the FET, or a call for a named record is made, while the index does not allow such, or a record number beyond the capacity of the index is given, or a record name is given for reading but not found in the index, or a new record name is given for writing but the index is full, or the file device is not allocatable, IORANDM will abort with a suitable message, or set error status in the FET and call error OWNCODE if any. See Error Procedures and Messages on page 14-121.

The question of whether a random file contains named records or not depends on how the record was addressed the first time any record was written randomly on the file. The first word of the index is zero when the index is empty. If the first writing of a record is by name, the first word is made negative; if by number, positive non-zero. In the latter case, writing or reading of records by name will not be allowed. This is because two words per record are needed if by name, but only one if by number; to make the latter saving of space possible, it is necessary to decide one way or the other the first time the index is used.

Other Programs Called

IO, CPC.

Registers Destroyed

A1, A6, X1, X6.

Narrative

IORR begins by calling subroutine IOSAV (in program IO) with 2 in the calling sequence to show that the calling sequence to IORR was 3 words long. IOSAV will save all the registers except A1, A6, A7, X1, X6, X7. Set up the final exit from IORR at IORE, in program IO; sets B1=1, A0=the address of the

first word of the FET, B3 = the address of the first word of the workspace, B2 = the address of the last+1 word of the workspace. Then we call subroutine IORA to find the index pointers in the FET, put the starting address of the index in B5, its length in B6, and the number of the last-addressed record in B7, and to wait until the FET shows non-busy status. Then set OUT=IN in the FET.

Next call subroutine IORB to convert the record name to a number, if necessary, and then to leave the record number in X1 and in cell IORN and the corresponding disk address from the index in X5. Now if X1 = 0, we had a name that was not in the index, and we go to IORER5A. Otherwise, set X0 = 10B to indicate reading (this will be used after we join program IO at IOIO). Now we load X1 with the record number from IORN (this seems completely unnecessary since it already contains the record number). If X5 = 0 there is no disk address for the record; hence it has never been written, and we go to IORER5A.

Next put the record number in bits 48-59 of the 8th word of the FET, as "Last record number addressed". Then put the disk address from the index into bits 0-23 of the 7th word of the FET, where it will control reading the next time CIO is called.

Finally, branch to IOIO in program IO. Assuming that all the checks have been good, the only difference between approaching IOIO from IORR and approaching it from IOREAD is that between IORR and IOIO we have set the buffer to empty, found a disk address in the index, and put it in the 7th word of the FET. This disk address field of the FET is otherwise 0, because the PP program that uses it if non-zero, zeroes it immediately after using it.

In writing we begin at IORW by calling subroutine IOSAV in program IO, just as at IORR. Then call subroutine IORA, just as at IORR. Now see whether the present status of the FET is write, and if so call CPC to write the end of the record. Then set the FET status to write completed, so that the safeguard in the preceding sentence will operate for the record we are about to begin writing, even if it turns out to contain no words of information.

Next call subroutine IORB, from which we return with the record number in X1, or if X1=0 because the record is addressed by name and the name is not in the index, then X4 contains the number of the first empty slot in the index, and cell IORN contains the record name. So if X1 is not 0, we go straight to IORWB. Otherwise, if X4 = 0, there is no room in the index for a new record and we go to IORER6A. If X4 is not 0, we shall put it in X1 to use as the number of the new record. But first store the record name from IORN at the word which is twice the record number oafter the first word of the index, minus 1; then set X1=X4, and arrive at IORWB.

At IORWB, we look at the first word of the index. If it is negative, the index contains names, and we set X2 to point to the word twice the record number past the first word of the index. If positive, the index is for numbered records only, and we set X2 to point to the word once the record number past the first word of the index. At IORWC, set X5=X1, the address

of the index word into which the disk address of the record must be stored when it begins to be physically written. Then branch to IORRA, where we put the record number (still in X1) in bits 48-59 of the 8th word of the FET; and put the address of the index word in bits 0-23 of the 7th word of the FET. Then branch to IOIO in program IO. Assuming that all the checks have been good, the only difference between approaching IOIO from IORW and approaching it from IOWRITE is that IORW has ended the previous record if the status was write; has then set the status to write completed; has perhaps put a new record name in the index; and has put an address in the FET so that when the first PRU of the record is written, its disk address will be properly stored in the index. At the same time, the PP program will zero that word of the FET to prevent a further storage in that index word.

Error Procedures and Messages

1. If the FET when first checked shows the file is on non-allocatable equipment, we abort the run through IORER7 with the message "FILE DEVICE NOT ALLOCATABLE".

2. If the FET when first checked shows the file is not random, we abort the run through IORER1 with the message "RANDOM CALL, NONRANDOM FILE".

3. If there is no index pointer in the FET, or if the FET indicates a zero-length index, we abort the run through IORER2 with the message "NO INDEX POINTER IN FET, OR 0 LGTH. INDEX".

4. If the record is referencedby name, but the first word of the index is positive non-zero, showing that only reference by number is allowed, we abort the run through IORER3 with the message "FILE RECORDS NOT NAMED".

5. If the record is referenced by number, but the number is too large for the length of the index, we go to IORER4A. If the EP bit in the FET is not set, we abort through IORER4 with the message "RECORD NUMBER TOO HIGH". If the EP bit is set, we set bits 9-13 of the first word of the FET to 25B. Then if the error OWNCODE field of the FET is 0, we return to the calling program with the first word of the FET in X1. This is similar to an end-of-record return to the calling program (as the word must be positive, its first character being a letter), but can be distinguished from it because an end-of-record return is with the address of a word in the workspace in X1, which would make the left 42 bits all 0. If the error OWNCODE field of the FET is non-zero, we enter the OWNCODE as a subroutine with the first word of the FET in X1, and on exit from it return to the user program that called IORANDM with the same word in X1. This differs a little from the way OWNCODE is ordinarily called from CPC. In CPC, we transfer the exit word of subroutine CPC to the exit word of the OWNCODE subroutine; then set the exit word of CPC to be a branch to the second word of the OWNCODE; then follow the ordinary exit procedures of CPC, which restores registers and then jump into the OWNCODE.

6. If the call to IORANDM is for reading, and the record is named with a name not in the index, or is identified by number but has never been written, we follow the procedure of the preceding paragraph, substituting IORER5A and IORER5 for IORER4A and IORER4, and error status 26B for 25B, and using message "RECORD NAME NOT IN FILE INDEX" in case of an abort.

7. If the record is to be written by name and the index allows names but contains neither the wanted name nor a free slot, follow the same procedure using IORER6A and IORER6, error status 27B, and message "NO ROOM IN INDEX FOR NEW NAME" in case of an abort.

Subroutines

IORA

This is called to check the FET of a random file to find its index, and to set up registers as shown below in "Exit Information", and to wait until the FET shows a not busy status. If the FET shows that the file belongs to a non-allocatable device, or is not random, or has no index or a zero-length index, we abort with a message. If the device is non-allocatable, we also zero the random bit in the FET.

Note that before calling subroutine CPC02 to wait for non-busy FET status, we have to save B2 in X3, and then set B2=A0. After CPC02 returns, reverse the process. This is because in program CPC, B2 is in general an address of the FET. While in IO and IORANDM, B2 is in general the last+1 address of the workspace. In IO, CPC is called at CPC, so that all registers are saved, reloaded, and restored afterwards. But here we are calling a little subroutine in CPC so that the saving, reloading, and restoring is not automatic.

Entry Information

B1 = 1, A0 = address of first word of FET.

Exit Information

B5 = starting address of index, B6 = its length, B7 = number of last - addressed record.

Subroutines Called

CPC02.

Registers Destroyed

A1, A3, A6, X0, X1, X2, X3, X6.

IORB

This is called after subroutine IORA. If the record is addressed as number 0, we add 1 to the last-addressed record number (which IORA left in B7) and use that as the record number. Then put the record name or number in IORN. Now if a name, we must convert it to a number. If already a number, branch to IORBQ, where if the first word of the index is zero, we set it positive non-zero to forbid named records in future. Find the word that contains the disk address of the record, but go to IORER4A if the record number would place that word beyond the end of the index. Then put the record number in X1 and the disk address (which we shall not after all use, if writing), in X5, and go to IORBV. There we zero the code and status field of the first word of the FET, except for the completion and mode bits, so that an end-of-record status will not prevent CPC from starting to read a new record, if we are reading. Then exit from IORB.

If the record was named, we have to search the index. Preset X1 = 0, to be the number of the record if the name is found, and X4 = 0, to be the number of the first unused slot in the index. Now if the first word of the index is zero, showing it has never been used, set that word negative to show named records are allowed. Then, if the first word of the index is positive, names are not allowed, and we abort. Then we scan the index for the wanted name. If found, we have the corresponding number in X1; we save it at IORN, keep the corresponding disk address in X5, and go to IORBV as above. If the name is not in the index, we exit from IORB with 0 in the X1, and the number of the first empty slot in X4, or 0 in the X4 if the index is full. In this case we do not clear the code and status field, nor replace the record name in IORN by its number. But these changes are actually necessary only for reading, and in reading if we do not find the name in the index, we take an error exit.

Entry Information

B1 = 1, A0 = address of first word of FET; B5 = address of first word of index; B6 = length of index; B7 = number of last-addressed record; X4 = address of the first parameter in the calling sequence to IORR or IORW (this was set by subroutine IOSAV).

Exit Information

The record number in X1, or X1 = 0 if the record is named and the name is not in the index; if the record is named and X1 = 0, X4 will contain the number of the first vacant slot in the index, or 0 if there is no vacant slot; unless the record was named and the name could not be matched, X5 = the present disk address of the record, as taken from the index.

Subroutines Called

None.

Registers Destroyed

A2, A3, A5, A6, X2, X3, X6, B4, B6.

14.16 RETURN

RETURN, lfn1, lfn2, . . .lfnn.

RETURN is a central memory program written for SCOPE 3.1.6. It is called by a RETURN control card.

Upon entry, RETURN expects file names (lfn1, lfn2, . . . lfnn.) to be placed in central memory words RA+I (I = 2, . . . 53B) and parameter count in word RA+64B by 1AJ. It then performs a close-unload (CIO) on each file in the list while decrementing parameter count to zero before it exits. Word RA+64B is unaltered.

RETURN card without any file name in its list will create zero-valued parameter count in word RA+64B and thereby cause an error message

▽RETURN CARD WITH NO PARAMETERS IS ILLEGAL▽

to be displayed in the B display and sent to the job and system dayfiles.

14.17 REWIND

This routine has two entry points, REWIND and UNLOAD. The following procedure takes place:

1. Depending on the entry point entered, the CIO function code (MODE) is set as follows:

   MODE = 60B for UNLOAD
   MODE = 50B for REWIND

2. The file is not opened.

3. The CIO call for REWIND/UNLOAD is issued.

4. An END request is issued to terminate processing.

14.18 UNLOAD

See above under "REWIND".

14.19 UPDATE

IMPLEMENTATION

The following discussion assumes that the reader is familiar with the characteristics and features of UPDATE as outlined in the SCOPE 3.1.6 Reference Manual.

The program library, which UPDATE creates and/or modifies, contains the symbolic data for programs being maintained on the system tape. These programs are arbitrarily divided into decks by insertion of *DECK and *COMDECK cards into the text stream. The program library file consists of a directory, a deck list, and text stream.

UPDATE may be used to create a program library (creation process) or modify an existing program library (correction process). UPDATE determines the process desired by interrogating the first card in the input stream. If the card is a *DECK or *COMDECK, a creation process is being instituted. Any other control card indicates a correction process.

CREATION PROCESS

The subroutine NEW copies the input stream from the input unit(s) onto the file CMPSCR while extracting deck anmes and common deck names. These names are added to the correction directory, DIRECT, and also to the history deck list, DECKS. When the end-of-record is encountered, UPDATE rewinds CMPSCR and generates a new program library and compile file. The basic subroutines involved here are WRNPL (write new program library) and WRCOM (write compile). The CHB's for the new cards are created for each new deck.

CORRECTION PROCESS

The correction process requires two logical passes. The first pass, CORR, verifies the correctness of the instructions given to UPDATE. CORR reads the entire input stream, constructs a dictionary (DICT) of the requested operations, and saves the insertion text in the buffer FPAGE. FPAGE overflows to disk file, FTEXT, when it exceeds its maximum size. The second pass, ECOR, generates an updated new program library, NEWPL, a compile file and a source file. ECOR reads the old program library, checking each card read against requested operations as saved in the dictionary. When a match is found the card is activated, deactivated, or text insertion is initiated as indicated.

CORR

A correction set is defined as being that set of cards which starts with an *IDENT card, *ADDFILE card, or *PURGE card, and terminates with one of thos cards or the end-of-record.

When a *PURGE card is encountered, the program searches the table DIRECT for the identifier, sets the purge bit (bit 16) and blanks the name (. . . .).

When an *IDENT card is encountered, it is verified that this is not a duplicate identifier, and the correction set is read in. Corrective operators *INSERT, *DELETE, *YANK, or *RESTORE cause entries to be made in the correction dictionary, DICT. Yank cards are also placed in FPAGE as text information.

When an *ADDFILE card is encountered, the file named is read to extract the new deck names and/or common deck names. These names are checked to make sure they do not already appear in the table DIRECT. If not, the names are entered in the tables DIRECT and DECK. The text is not put in DICT, rather a dummy insert, indicating an ADDFILE. At the end of the first pass, the contents of FPAGE (text information) is dumped onto file FTEXT.

<u>ECOR</u>

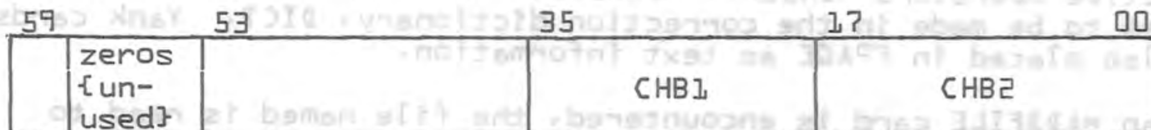
A counter is set up for each ident in the DIRECT table. As card images associated with an ident are processed, the current card counter for the ident is bumped. The counters are kept in the table, CNTR.

ECOR searches the DICT table for any insertion entry whose card number matches the current card number of the ident being referenced. If an insert is found to apply at a current card counter, the text associated is retrieved from FTEXT. If the insert is a dummy, indicating an ADDFILE, the text is read from the appropriate file. If no insert applies at any of the current counters a card is read from the old program library and the appropriate card counter is bumped by one. In any case, a card has been obtained to process.

The DICT is now searched for a delete or restore entry which applies to the card being processed. If found, a new CHB entry indicating activate or inactivate is placed in the CHBTBL. If the card in question is at the end of a delete range, the DICT entry is changed to an insert in case the delete had cards following. Searching continues for further delete or restore cards applying to the same card. When the search fails (end of DICT reached) YANK and PURGE are looked for. YANK causes the yank bit to be set in the CHB. PURGE causes the CHB entry to be removed. The activity of the card is determined by a CHB search. If the activity has changed the card is printed. If active, the card is written on the New Program Library, compile file, and source file. The appropriate card counter is bumped and the search for inserts is reinitiated.

TEXT STREAM

The text stream contains card images and control information known as correction history bytes (CHB's). At least one word of control information precedes each card image:



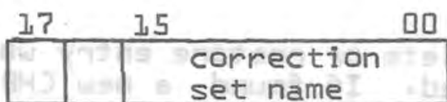
Secondary CHB's recorded in bits 17-00 of the first word continue in subsequent 18-bit bytes in the lower order 54-bits of each word and terminate with a zero-value CHB.

Primary CHB; this byte identifies deck or correction set under which this card was introduced and gives the card its alphanumeric name.

Number of words used by the compressed card image; this information speeds up input operations.

Activity bit; contains a 1 if the text card, which immediately follows the control information, was active at the time the program library is written.

CHB format:



identifies correction set which performed action; provides ordinal into identifier table.

Activity bit; contains a 1 if the correction set activated this card.

Not used.

TABLE FORMATS

DYNAMIC TABLES

DIRECT: 1 word per entry.

59 ... 18	17	16	... 0
7 character ident name	Y	P	

Y = Yank bit, on if ident has been yanked.

P = Purge bit, on if ordinal has been purged (in which case, name = ...).

The DIRECT table contains an entry for each IDENT encountered by UPDATE; this is, from the input stream or the old program library.

DECK: 1 word per entry.

59 ... 18	17 ... 0
7 char deck name	

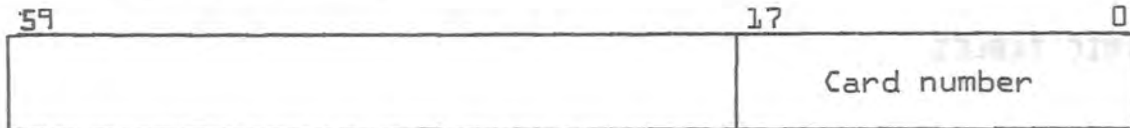
DICT: 3 words per entry.

59	57	47 ... 30	30 ... 17	17 ... 0
		Index of 1st ident of limit		Card num of ident
		Index of 2nd ident of limit		Card num of ident

59 ... 53	53 ... 36	36 ... 17	17 ... 0
	Index of ident corr under	Number of cards to insert	Address of text on FTEXT

If a restore or delete has only one limit, the 2nd word is a copy of the first. For an insert the second word is meaningless. For an ADDFILE the second word contains the file name left justified and word three is set to 1000000.

CNTR: 1 word per entry.



This table contains the count of cards found for each ident. The count for the Ith ident in the DIRECT table is found at location CNTR + I.

FIXED LENGTH TABLES

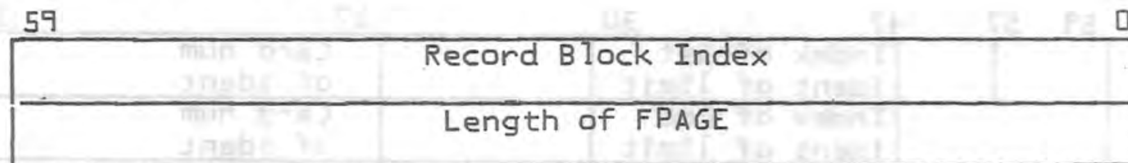
CHBTAB: 1 word per CHB.

Y = Yank bit

A = Activity bit

The CHBTAB can hold 200 CHB's. The lower 18 bits have the same format as the 18 CHB's which are written on the OLDPL with the only difference being that bit 17 is used to indicate that an ident has been yanked.

FTEXT: 2 words per entry.



UPDATE

ENTRY

From SCOPE loader

ROUTINES CALLED

ADDWORD, READCD, CLASSIFY, NEW, RBIN, MANAGER, CORR, ECOR

DESCRIPTION

UPDATE is the initialization routine. It scans the UPDATE control card and sets up all the optional flags and resets file names. It opens files which are always present and initializes the dynamic tables. The first card is read to determine the style of updating. If the input file is empty, ECOR, if not either NEW or CORR is called.

ECOR

ENTRY

EQ ECOR

ROUTINES CALLED

CORIDX, DUMTXT, CONDEC, PRINT, DUMDIR, ROPL, MOVE, MANAGER GETTXT, RDEC, SQUEEZE, UCARD, ADDID, CKOVL, WRNPL, WRCOM, SCITEM, COPYM, WRSOUE

DESCRIPTION

ECOR is called from CORR to read and make the corrections called for in the table DICT. Modifications are listed and overlapping corrections checked for. The NEWPL, SOURCE, and COMPILE files are created by calls to WRNPL and WRCOM. At the end either DONE or ABORT is called to terminate the UPDATE run.

WRCOM

ENTRY

RJ WRCOM

ROUTINES CALLED

CLASSIFY, WRSOU, WRC, WMC, WBIN, UCARD, ADDID, PRINT, SCITEM, TLUDIR, GETCH, COPYM, ADDWORD, MOVE, WRSOUE

DESCRIPTION

WRCOM is called by ECOR and NEW to write the compile and source files. It also determines which file to write cards to, and sets the yank bit in the directory for yank cards. Decks written to the compile file and comdecks encountered are also tabulated.

NEW

ENTRY

EQ NEW

ROUTINES CALLED

READCD, CLASSIFY, WMC, SCITEM, ADDWORD, ADECK, PRINT, DUMDIR, RMC, ADDID, WRNPL, WRCOM

DESCRIPTION

NEW is the section of UPDATE which handles creation runs. Text cards are read in and copied to the CMPSCR file. Deck names are extracted and kept. At the end of the input stream CMPSCR is rewound and the routines WRNPL and WRCOM are called to create the NEWPL, SOURCE and COMPILE files.

CORIDX

ENTRY

RJ CORIDX

ROUTINES CALLED

ADDWORD

DESCRIPTION

CORIDX checks to see if an ident is in effect, and if so, adds the ident name to the table DIRECT.

GETLIM

ENTRY

RJ GETLIM

EXIT

X6 contains ordinal of ident
X7 contains the card number

ROUTINES CALLED

SCITEM, TLUDIR, SCNN, PRINT, GETCH

DESCRIPTION

GETLIM scans a control card for an ident and card number and returns this information to the calling routine. Format errors and a card number of zero are diagnosed.

CLASSIFY

ENTRY

RJ CLASSIFY

In call RJ is followed by a list of words. The first 30 bits of each contain an EQ jump to a processing routine; the second 30 the address of the control word to be matched. The list terminates with a word of 0's.

ROUTINES CALLED

SCITEML, GETCH

DESCRIPTION

CLASSIFY compares the card image with the string of types and takes the proper jump for a match. If none match the card, the word following the zero word is returned to.

WRC

ENTRY

RJ WRC

ROUTINES CALLED

FMCARD, CLASSIFY, WDEC, ADDWORD, SCNN, PRINT, SCITEM, RBIN, MANAGER, MOVE

DESCRIPTION

WRC formats and writes the data card to the COMPILE file. It finds comdecks called for on CALL cards and copies them to the compile file.

CKOVL

ENTRY

RJ CKOVL

DESCRIPTION

CKOVL checks to see if the overlap flag has been set for a card. If the flag has been set, the word overlay is appended to the card image. Overlapping corrections are also displayed on the scope.

READCD

ENTRY

RJ READCD

EXIT

X1 = 0 normal
X1 ≠ 0 if EOR encountered

ROUTINES CALLED

RDEC, SQUEEZE, MOVE, CLASSIFY, PRINT, ATTACH, SCNN

DESCRIPTION

READCD handles the input of coded information for UPDATE. The only exception is for files named by ADDFILE. Cards are read, compressed, and checked to see if they are file manipulation cards. Comment cards are printed out and another card read automatically.

PRINT

ENTRY

RJ PRINT
X1=FWA to be printed
X2=word count 13 words

ROUTINES CALLED

WDEC, MOVE

DESCRIPTION

PRINT handles all output from UPDATE to the list file. Page counts are also handled.

GETCH

ENTRY

RJ GETCH

EXIT

Character in X6

DESCRIPTION

GETCH advances the column pointers and fetches the next character of the card.

SCITEM or SCITEML

ENTRY

RJ SCITEM or RJ SCITEML

EXIT

X6=identifier LJUST with trailing 0's.

ROUTINES CALLED

GETCH

DESCRIPTION

SCITEM and SCITEML both reply with the ident in X6. SCITEM, however, checks for an identifier greater than 7 characters where SCITEML doesn't.

SCNN

ENTRY

X6=value of field or 0 if illegal

ROUTINES USED

GETCH

DESCRIPTION

SCNN scans a numeric field on a card and returns the integer value.

CONDEC

ENTRY

RJ CONDEC
X1=integer to be converted

EXIT

X6=display form leading blanks, B6=6 times count of digits

DESCRIPTION

CONDEC converts an integer to display code for printing purposes.

DUML

ENTRY

RJ DUML
X1=origin of table
X2=length of table

ROUTINES CALLED

PRINT, CONDEC

DESCRIPTION

DUML dumps a directory, 6 entries to a line, by formatting a line and calling PRINT. Entries in a directory of . . . . . are considered purge idents and are counted.

COPYM

ENTRY

RJ COPYM

ROUTINES

MOVE, ADDWORD, CLASSIFY, RMC, WRC

DESCRIPTION

COPYM is called by the write compile routine to handle its I/O. It determines whether or not the card is part of a comdeck, and if the card should be written to the compile file or CMPSCR file.

DUMTXT

ENTRY

RJ DUMTXT

ROUTINES CALLED

MANAGER, ADDWORD

DESCRIPTION

DUMTXT writes a page from memory to the random file FTEXT. It adds to the dynamic table TXTLIM both the random index of a left record and its length. It also keeps track of the total number of words written, TXTBIAS.

GETTXT

ENTRY

RJ GETTXT

DESCRIPTION

GETTXT fetchs the right indexed record from the file FTEXT.

MANAGER

ENTRY

RJ MANAGER
A0=table to be managed
X1=size of change (+ or - words)

EXIT

X2=origin of table
X3=new size

ROUTINES CALLED

MANAGD

DESCRIPTION

MANAGER is used to allocate space to UPDATE's dynamic tables. If room does not exist to make the change, the present lengths of tables are summed, the difference of the sum and what's available calculated, and this difference divided into equal increments for each table. If not enough room exists, the job is aborted.

MANAGD

ENTRY

RJ MANAGD

ROUTINES CALLED

MOVE

DESCRIPTION

MANAGD moves all of UPDATE's dynamic tables to low core.

SQUEEZE

ENTRY

RJ SQUEEZE

DESCRIPTION

SQUEEZE compresses the card image and determines the length of the compressed image. It terminates the image with a 0000_8 byte.

UCARD

ENTRY

RJ UCARD
X1=add. of first word

EXIT

X6=add of word following last word unpacked.

DESCRIPTION

UCARD unpacks the compressed card image from SQUEEZE and places it in the array card. It terminates when it finds a word ending with a 0000_8 byte.

ADDID

ENTRY

RJ ADDID

ROUTINES CALLED

CONDEC

DESCRIPTION

ADDID picks up ident name and card number of a text card and adds the information to the cells CARD+7 and CARD+8.

ROPL

ENTRY

RJ ROPL

ROUTINES CALLED

RBIN

DESCRIPTION

ROPL reads the old program library, unpacking CHB's and checking for premature EOR's. It also bumps the CNTR for the ident which introduced a card.

ADECK

ENTRY

RJ ADECK
X1=DECK name to be added to DKLST

ROUTINES CALLED

ADDWORD

DESCRIPTION

ADECK adds a deck name to the dynamic table DKLST.

ATTACH

ENTRY

RJ ATTACH

ROUTINES CALLED

SCITEML, PRINT

DESCRIPTION

ATTACH is called to attach a named file to the FET for READFIL. It checks the legality of the file name before attaching the file.

ENTDICT

ENTRY

RJ ENTDICT

ROUTINES CALLED

ADDWORD

DESCRIPTION

ENTDICT builds a three word dictionary entry in the table DICT. The entry has the following format:

Wrd	Bits	
1	59-57	1-insert 2-restore 4-delete
	47-30	Index of ident of card
	17-0	card number
2	47-30	index of ident of second limit
	17-0	card number of second limit
3	53-36	index of ident this correction is under
	35-18	number of text cards to be inserted
	17-0	address of start of text within FPAGE

FMCARD

ENTRY

RJ FMCARD

ROUTINES CALLED

UCARD, CONDEC

DESCRIPTION

FMCARD unpacks the squeezed image and appends the ident and card number to it.

WRSOUE

ENTRY

RJ WRSOUE

ROUTINES CALLED

WDEC

DESCRIPTION

WRSOUE checks if an *END card is needed on the source file and writes one if needed.

WRSOU

ENTRY

RJ WRSOU

ROUTINES CALLED

UCARD, WDEC

DESCRIPTION

Unpacks card and writes it to the source file.

RBIN

ENTRY

RJ RBIN
A1=FET loc of file to be read
X2=FWA of data
X3=number of words to be read

EXIT

X1=0 if normal exit
X1≠0 if EOR encountered

DESCRIPTION

RBIN moves the number of words specified from the file's input buffer. If another read is necessary to satisfy the request, it is issued.

RDEC

ENTRY

RJ DEC
A1=FET loc of file to be read
X2=FWA of data
X3=max number of words to read

EXIT

X1=0 if normal exit
X1≠0 if EOR encountered

DESCRIPTION

Similar to RBIN except RDEC terminates moving when it encounters a word terminating with a zero byte. If X3 is met without finding a 0 byte, pointers are advanced without moving any words.

WBIN

ENTRY

EJ WBIN
A1=FET loc of file to be written
X2=FWA of data
X3=number of words to be written

DESCRIPTION

WBIN moves data to the file's circular buffer and initiates a write if it is necessary.

WDEC

ENTRY

RJ WDEC
A1=FET loc
X2=FWA of data
X3=number of words to write

ROUTINES CALLED

WBIN

DESCRIPTION

WDEC writes data by calling WBIN. It makes sure that the string written terminates with a 0 byte (12 bits).

WMC

ENTRY

RJ WMC

ROUTINES CALLED

WBIN

DESCRIPTION

Writes a squeezed card image and its length to CMPSCR by calling WBIN.

RMC

ENTRY

RJ RMC

EXIT

X1≠0 if EOR encountered

ROUTINES CALLED

RBIN

DESCRIPTION

Read information written on CMPSCR by WMC.

WRNPL

ENTRY

RJ WRNPL

ROUTINES CALLED

WBIN

DESCRIPTION

WRNPL writes the text information and the CHB's to the NEWPL by calling WBIN. It packs the CHB's from the CHB table before writing them.

LJUST

ENTRY

RJ LJUST
X1=name to be justified

EXIT

X6=name blank filled
X7=name 0 filled

DESCRIPTION

LJUST takes the name input and left justifies it blank filling and 00 filling it.

TLUDIR

ENTRY

RJ TLUDIR
X1=name to search for

EXIT

X1=same as on entry
X2=0 if name not found
=actual entry found
A2=address of entry
X3=index of entry

DESCRIPTION

TLUDIR searches the ident directory for the name input.

ADDWORD

ENTRY

RJ ADDWORD
X1=word to be added
A0=number of table to be added to

ROUTINES CALLED

MANAGER

DESCRIPTION

ADDWORD adds a data word to a dynamic table.

DUMDIR

ENTRY

RJ DUMDIR

ROUTINES CALLED

WBIN, DUML, MANAGER

DESCRIPTION

DUMDIR opens the NEWPL and dumps the ident and deck list to it. It also prints the two lists and sets up the counters for the card numbers. DUMDIR may be entered only once.

MOVE

ENTRY

RJ MOVE
X1=number of words to be moved
X2=source address
X3=destination address

DESCRIPTION

MOVE moves blocks of words in storage. The direction of the move is checked to avoid over-stores.

DONE

ENTRY

EQ DONE

ROUTINES CALLED

DUML, PRINT

DESCRIPTION

DONE is the normal termination routine. It terminates UPDATE's files, lists out common decks and decks written to the compile file, and prints statistics about the UPDATE run.

ABORT

ENTRY

EQ ABORT

ROUTINES CALLED

CONDEC, DUMDIR

DESCRIPTION

ABORT is the abnormal termination routine. It terminates

UPDATE's files, prints idents and decks encountered, displays number of errors found, and calls the system's abort routine, ABT.

CORR

ENTRY

EQ CORR

ROUTINES CALLED

PRINT, READCD, CLASSIFY, SCITEM, ADECK, MOVE, DUMTXT, CORIDX, TLUDIR, GETLIM, ENTDICT, GETCH, ATTACH, RDEC, SQUEEZE, ADDWORD

DESCRIPTION

CORR is the section of UPDATE which reads and checks correction sets. Cards are checked for illegalities and, if legal, build an entry in the table DICT. When all input has been read, CORR transfers control to ECOR.

14.20 XXXDMPQ

General

XXXDMPQ determines the type of file being dumped by interrogating the parameter in RA+2. Depending on the file type, the corresponding label is set in the FET. A retention code of one day is set into the label. Two buffers are used for dumping the files. A read is issued for one buffer and when complete, the name is changed and a write issued and a read is issued into the other buffer.

The first record of each file (one file per job) is three words long and contains the FNT entry for that file. The file is dumped until end-of-information is returned from 1SQ and at that time a file mark is written. When no more FNT entries exist, the file is closed (from the PP) and trailers labels written.

Major Subroutines

1. ENTRNAM

   This routine is called each time a new file is being processed. It moves the FNT entry into the I/O buffer and sets up a cell which contains the name of the file being dumped.

2. CALLPP

   This routine is called when a file is ready to be processed. It sets the cell that the PP is monitoring to zero which signals the PP to write an FNT entry. The message "looking for XXXX" is output from this routine. The message is displayed whenever the PP does not respond with an FNT entry.

3. READ1

   This routine sets up the specified FET for a read operation. It sets up the file name and function code in the FET, sets first = in = out and issues a read.

4. CKREAD1

   This routine checks the status of a completed read operation. It checks for an end-of-record or end-of-file status and checks if the buffer is full.

   If the status is EOF, a check is made to determine if end-of-information is up, if not, the record is written with an end-of-record write. If the status is not EOR,

the buffer must be full {IN = LIMIT-1}. If the buffer is not full, another read is issued with auto-recall and when complete the entire subroutine is executed again. The only way to exit the routine is for the status to be EOF, EOR or buffer full.

5. WRITE1

   This routine sets up the specified FET for a write operation. The name of the file is changed to "QUETAPE" and a write is issued to CIO.

6. CKWRITE

   This routine checks the status of a completed write operation. If an unrecovered write parity error has occurred, a jump to the routine PARER is made. If the buffer is empty, an exit is made, otherwise another write is issued.

7. PARER

   When an unrecovered write parity error occurs, the following procedure is taken.

   1. Back up tape to the beginning of the file.
   2. Rewind the disk file.
   3. The message "WPE UNRECOVERED, EOT, FORCED, TYPE GO" is issued to the dayfile.
   4. End-of-tape condition is forced and the tape rewound and unloaded.
   5. The message MTXXEOT, MTXX NOT READY is displayed.

8. WRITFIL

   This routine is used to write the file mark at the end of each file.

General Flow

It should be noted that the primary design criteria for DMPQ was speed of execution in minimal field length. Therefore, as much read/write overlap as possible was obtained. If greater overlap is attempted, I/O timing considerations make the routine unreliable.

The routine is loaded by loader and the appropriate tape label is set into the FET's. A request is then issued for the tape and the job waits in recall for operator assignment. Upon operator assignment, the file is open via OPE for write operations. XDQ is then called with a buffer address in the lower 18 bits of the call. A call is made

to CALLPP to obtain an FNT file and upon return, a call is made to ENTRNAM to move the file name into the I/O buffer, then a write is issued. A read is immediately issued from the disk for the first record of the file.

Error Conditions

The diagnostic "ERROR AT EOF0, NOTIFY SYSTEMS" is put out if end-of-information is encountered on the first read of a file. This condition cannot logically happen and is flagged in case it does. EOF0 is a location tag in XXXDMPØ.

14.21 XXXRESQ

General

XXRESQ reads the tape a record at a time and writes the information to the disk. The first record is saved (the FNT entry) and when a file mark is encountered on the tape, the FNT address of the file restored is set into a cell that is being monitored by XRQ. XRQ picks up the FNT entry and clears the cell and XXXRESQ continues reading the tape. Two buffers are used to provide continuous processing of the tape. When end-of-information is returned from the tape driver, the message "RESTORE COMPLETED" is written to the dayfile.

Major Subroutines

1. CALLPP

   This routine is entered when a file mark is encountered on the tape. A local cell is checked to determine if a parity error has occurred, if the flag is set, the message FILE NAME DELETED FROM QUEUE, PARITY ERROR" is displayed, "GO" must be entered to continue processing. Only "INPUT" type files are deleted because of parity errors. If the error flag is not set, the FNT address of the file being restored is set into the location monitored by XRQ.

2. WRITE

   This routine sets the disc file name into the FET and issues a write to CIO.

3. READ

   This routine sets the tape file name into the FET and issues a read to CIO.

4. SAVINFO

   This routine is called after the first read of every file. The first record contains the FNT entry. The FNT entry is moved and saved in local storage. The file name is masked out of the first word and saved in a cell tagged DISCNAM. A message containing the file name is sent to the dayfile.

5. EOI

   This routine is entered when end-of-information is returned

from the tape driver. The FNT address is processed in the same manner as CALLPP and the message ▽RESTORE COMPLETED▽ is displayed. Minus zero is set into the location monitored by the XRQ which signals the PP routine to drop the CP and drop the PP.

General Flow

Upon initial entry, the parameter from XRQ is interrogated and the correct tape label is set up on the FNT. A request is then issued for the tape. Upon operator assignment, a call is made to XRO with the address of a four word buffer in the lower 18 bits of the call. A read request is issued and on completion a call to SAVINFO is made. Tape reading and disk writing continue until end-of-file status is returned from the tape. At that time, CALLPP is called and then the process continues until end-of-information is returned from the tape. At that time, EOI is called to terminate the run.

Error Conditions

The diagnostic ▽TAPE FORMAT ERROR, NOTIFY SYSTEMS▽ is sent to the dayfile if end-of-file is returned on the second read of each file.

from the tape driver. The FNT address is processed in the same manner as CALLPP and the message "RESTORE COMPLETED" is displayed. Minus zero is set into the location monitored by the XRQ which signals the PP routine to drop the CP and drop the PP.

General Flow

Upon initial entry, the parameter from XRQ is interrogated and the correct tape label is set up on the FNT. A request is then issued for the tape. Upon operator assignment, a call is made to XR0 with the address of a four word buffer in the lower 18 bits of the call. A read request is issued and on completion a call to SAVINFO is made. Tape reading and disk writing continue until end-of-file status is returned from the tape. At that time, CALLPP is called and then the process continues until end-of-information is returned from the tape. At that time, EOI is called to terminate the run.

Error Conditions

The diagnostic "TAPE FORMAT ERROR, NOTIFY SYSTEMS" is sent to the dayfile if end-of-file is returned on the second read of each file.

14.22 TRANSF

TRANSF, jobname, ...

TRANSF is a central memory program written for job dependency in SCOPE 3.2.1. It is called by a TRANSF control card.

Upon entry, TRANSF expects job names (five characters or less) to be placed in central memory words RA+I (I = 2,...53B) and parameter count in word RA+64B by 1AJ. It then calls PP routine JDP to process all the job names one at a time.

A TRANSF card without job names is ignored.

15.0 COMPASS Debug Aids

15.1 SNAP

15.1.1 Introduction

SNAP, the snapshot dump routine, is executed in two phases. During phase one, the routine SNAP is entered by loader for the purpose of planting traps in the object program locations at which SNAPS are to be taken. In order to achieve this, LOADER (via DEBUG) provides the following information:

INDEX2 = The object routines entry point address (transfer address)

INDEX3 = FWA of the SNAP routine

INDEX4 = LWA+1 of the SNAP routine

INDEX5 = FWA of the LOADER provided snap tables. (SNAP table formats are shown in Fig. A)

SNAP checks the validity of the parameters supplied in the tables, and if they are acceptable, stores the contents of the object program location being snapped in the tables. SNAP then plants a return jump to the routine SNAPPER in the snapped location.

For each SNAP table in which there is an error, the dayfile message "SNAP CARD PARAMETER ERROR, ID = XXXXXXX" is issued and the table is ignored. The return jump to the routine SNAPPER is formatted as shown below:

Where address is the FWA-1 of the LOADER provided table associated with this snap. After all tables have been processed, SNAP enters the object routine at the address contained in the index register two.

Upon encountering, in the object routine, one of the SNAP planted return jumps, phase two (SNAPPER) is entered. The purpose of phase two is the

creation of the snapshot dump(s) in accordance with the specifications given in the loader provided tables, which were generated from the user's SNAP cards.

15.1.2 Subroutine descriptions

A. SNAPPER is a closed subroutine, and consists of a driver (main loop) and several other subroutines. SNAPPER performs the following functions:

1. Sets up a header line which appears at the top of each page of the dump.

2. Updates frequency parameters and determines if dumps are to occur.

3. Sets up, if requested to do so, the following call to a user supplied subroutine:

L	RJ	User Subroutine
L+1		Address 1
L+2		Address 2
L+3		Normal Return

Where: Address 1 = FWA-1 of the LOADER provided SNAP tables

Address 2 = FWA of the SNAPPER provided tables

Notes: 1. The user routine must return control to L+3

2. The user routine need not save registers

4. Determines mode and sets up a call to the proper subroutine to perform the requested dump.

5. Executes the object routine instructions which were replaced by the call to SNAPPER.

6. Returns control to the object routine.

B. MNEMONIC DUMP (MNEDMP)

The mnemonic dump subroutine dumps, the address of, and one 60 bit word of data per line of output. The dump is in COMPASS mnemonics and may include a maximum of four instructions per line of output. In order to

insure a high degree of accuracy the following checks are made:

1. If the op code is a thirty bit op code are there 30 bits remaining in the data word?
2. If the op code is zero, are the entire thirty bits of the instruction zero?
3. If the op code is 46 (no op.) are the remaining nine bits zero?
4. If the op code is 10, 14, or 47, are the j and k indicators the same?
5. If the op code is 01 and the sub-op code is not zero, is the instruction upper?

If any of the above tests fail a fifteen bit byte of data is dumped in octal.

Data is packed in a temporary buffer as it is processed. When enough data for one line of output has been created, the temporary buffer is reformatted and packed in the specified output buffer. After each word is dumped, the data FWA is increased by the increment and the dump continues until the FWA is greater or equal to the LWA, whereupon control returns to the main loop (SNAPPER).

C. INTEGER DUMP (INTDMP)

The integer dump subroutine treats all data as decimal integers of 60 bits. It dumps the first words address and up to four data words per line of output. The data to be converted is first floated and put in double precision floating point format. The log of the number (exponent) base 2 is converted to base 10. The number is then scaled and the exponent adjusted. The fractional portion of the number is converted to BCD by successive multiplication. The exponent is then used to determine the digit count. Data is rounded and packed in a temporary buffer

until enough data for a complete line has been processed. The data is then reformatted in the specified output buffer. Each number is left adjusted in its output columns. The method of conversion is similar to that employed by the FORTRAN object time routine KODER. FWA is incremented and the dump continues until LWA is equaled or exceeded by FWA.

D. DISPLAY DUMP (DSPDMP)

The display code dump subroutine considers all data input to be display coded characters. DSPDMP dumps the address of the first data word of each line and eight words of ten characters each for each line of output. Checks are made for end of line indicators, and if any are found they are replaced by an up and by a down arrow. FWA is incremented and the dump continues until LWA is equaled or exceeded by FWA.

E. FLOATING OR INTEGER DUMP (FOIDMP)

FOIDMP considers all data to be single precision floating point unless the exponent is zero (bits 48 through 59), in which case it considers it to be a 48 bit integer. The data is floated if required, and put in double precision floating point format. This value is then converted to display code as described in part C above. The FWA and four data words per line are output to the specified buffer. Floating point values are output in scientific notation with an E indicator if the value of the exponent is less than 100, otherwise the E is omitted. Fifteen digits are printed for all floating point values (no suppression of trailing zeros). Integer values contain a maximum of fifteen digits and are left adjusted in their output fields. FWA is incremented and the dump continues until FWA equals or exceeds LWA.

F. SINGLE PRECISION FLOATING POINT DUMP (SPFDMP)

Same as E except no test for integer values is made.

G. DOUBLE PRECISION FLOATING POINT DUMP (DPFDMP)

DPFDMP considers all data input to be in double precision floating point format. The FWA and three double precision words of twenty nine digits are output for each line. The method of conversion is as described in C above.

Twice the increment is added to FWA and the dump continues until FWA equals or exceeds LWA.

H. OCTAL DUMP (OCTDMP)

All data input is assumed to be of octal format. The data FWA and four words of twenty characters are dumped for each line of output. Each word is broken into four columns of five characters each. Each five character column is separated by a blank column. Successive repititious data are not suppressed. FWA is incremented, and the dump continues until FWA equals or exceeds LWA.

I. REGISTER DUMP (REGDMP)

The register dump routine dumps the contents of all twenty four operating registers as they appeared upon entry to SNAPPER. It also dumps the contents of the location contained in each of the address registers A1 through A7.

15.1.3 TABLE FORMATS AND PROCESSING

SNAPPER uses two types of tables. The loader provided tables are semi-permanent, multiple entry tables. The SNAP provided tables are single entry dynamic tables.

A. The loader provided tables are used primarily to pass parameters from the SNAP card to the SNAP and SNAPPER subroutines. They will also contain any parameters for a user subroutine. Except for the updating of the frequency parameters, SNAPPER does not modify these tables. There is one loader provided table for each SNAP card associated with a

given job. Each table entry is a minimum of nine words, but may be longer due to user routine parameters on the SNAP card. Like all loader tables, the SNAP tables are stored in descending order from high numbered core to low numbered core (backwards).

B. The SNAP provided tables provide information to and communicates with any user routine requesting entry from SNAP. The SNAP provided table is 26 words long and has the following symbolic locations defined as entry points:

1. <u>RBO</u> Is the FWA of an eight word array of the saved index registers.
2. <u>RAO</u> Is the FWA on an eight word array of the saved address registers.
3. <u>RXO</u> Is the FWA of an eight word array of the saved X registers.

All registers are stored in ascending order within these arrays. The remaining two words of the table are used by SNAP entered user subroutines to communicate with SNAP as follows:

1. If bit 59 of word RBO+24 is set to one, this SNAP will be suppressed.
2. If bit 59 of location RBO+24 is clear, an 18 bit address, low order, will be interpreted as the address of a 13 word FET which is to replace SNACE, as the output buffer for this SNAP.
3. An 18 bit address low order in location RBO+25 will be interpreted as an exit address to which control will be given upon completion of the snapshot dump.

Prior to entering a user routine, SNAPPER sets location RBO+24 to zero and location RBO+25 is set to the address of the location at which the snap is planted plus one (IA+1 for the current dump). Data in this table is pertinent to the present dump only.

76	Unused (18)		Pitch Down (18)		Pitch Up (18)
IDENT (42)					Unused
CIA					
Snap Count (15)				Step Size (12)	Mode (12)
A	Ovly (12)	F1 (15)		F2 (15)	F3 (15)
Unused					IA (18)
Unused					UR (18)
Unused					FWA (18)
Unused					LWA (18)
First User Param ⋮ Last User Param					
-0 (not present if no user params)					
76	Unused (18)		Pitch Down (18)		Pitch Up (18)

Fig. A LOADER PROVIDED SNAP TABLE

Where: Pitch Up = the increment to be SUBTRACTED from the current table FWA to reach the next higher table entry.

Pitch Down = the increment to be ADDED to the current table FWA to reach the next lower table entry.

Note: SNAP always processes tables upward, i.e., from high numbered core to low numbered core.

IDENT = A 7 char identification for this dump.

Note: If loader sets this word to zero, SNAP will skip the table. This is a device for use by loader to delete improper tables without aborting the job.

CIA = The contents of user location IA prior to planting the SNAP call.

SNAP COUNT = A fifteen bit count of the number of times SNAPPER has been entered due to the call planted at this table's IA.

STEP SIZE= The increment between words to be dumped. This value is doubled by SNAPPER for double precision dumps.

MODE = The dump mode indicator(s). Legal modes are as follows:

Ø = octal format

M = mnemonic format

S = single precision floating point format

F = single precision floating point or integer format.

C = display code format

I = sixty bit integer format

D = double precision floating point format

R = register dump

The R indicator may be used in combination with the other mode indicators. If no mode is given R & Ø are assumed. If, however, no mode and no FWA and LWA are given, only R is assumed. Illegal modes are ignored.

A = A two bit overlay/segment flag. If A=2, this SNAP is associated with overlays. If A=3, this SNAP is associated with segments.

OVERLAY LEVEL= Overlay level to which this SNAP table applies. This value is compared to the value set in location OVFLAG (an entry point in the routine DEBUG) which is the level currently loaded. If during overlay processing, these two values do not match, the table is skipped.

F1 = Starting SNAP count. Dumping begins when the SNAP count is equal to F1.

F2 = Ending SNAP count. Dumping stops when the SNAP count is greater than F2.

F3 = The frequency increment. This value is added to F1 to determine when the next dump will be executed.

IA = SNAP address. This is the location at which the user wishes to execute a SNAP dump. The contents of IA are saved prior to planting the call to SNAPPER, and are executed in SNAP after restoring registers, but prior to returning to the object routine.

UR = User subroutine address. The address of a closed subroutine which, if present, will be entered by SNAP after updating the frequency parameters, but prior to any dump. This subroutine, if required, must be supplied by a user.

FWA = First word address of the area which is to be dumped.

LWA = Last word address of the area which is to be dumped.

Following the above will be any user parameters from the snap card. The user parameters are terminated by a word of all ones (minus zero). The word of ones is included only if user parameters are present.

Entry point	Register	Contents
ENTRY POINT RBO	B0	0
	B1	UPON ENTRY
	B2	UPON ENTRY
	B3	UPON ENTRY
	B4	UPON ENTRY
	B5	UPON ENTRY
	B6	UPON ENTRY
	B7	UPON ENTRY
ENTRY POINT RAO	A0	UPON ENTRY
	A1	UPON ENTRY
	A2	UPON ENTRY
	A3	UPON ENTRY
	A4	UPON ENTRY
	A5	UPON ENTRY
	A6	UPON ENTRY
	A7	UPON ENTRY
ENTRY POINT RXO	X0	UPON ENTRY
	X1	UPON ENTRY
	X2	UPON ENTRY
	X3	UPON ENTRY
	X4	UPON ENTRY
	X5	UPON ENTRY
	X6	UPON ENTRY
	X7	UPON ENTRY
	USER COMM.	WRD. 1
	USER COMM.	WRD. 2

Note: This table is in ascending order from low numbered to high numbered core.

Fig. B SNAP Provided Table

15.1.4 FLAGS, BUFFERS & TABLES

The following flags are used:

IFLG = Integer conversion flag, zero is on, non-zero is off.
Used by: FLTDSP. Set by: INTDMP, CTBD, & DPFDMP.

FSFLG = Floating/single or single only flag.
+1 = single precision floating point format only.
-1 = single precision or integer dump.
Used by: CTBD. Set by: FOIDMP & SPFDMP.

NODIG = Number of digits to convert to display code.
Used by: FLTDSP. Set by INTDMP, CTBD, & DPFDMP.

FWAFET = First word address of the FET to be used with this dump, normally SNACE.
Used by: PUTOUT & SCIO. Set by: SNAPPER.

LINECNT = Minus the number of lines to be printed per page.
Used by: PUTOUT. Set by: SNAPPER, PUTOUT.

WRDS = Number of data words to convert per line of output, except as used by MNEDMP where it is the number of 15 bit bytes processed in the current data word.
Set and used by all dump routines except REGDMP.

WRDS+1 = An overflow word which is destroyed each time address is executed.

OUTPUT = Used as a working buffer.

SDTA = Temporary storage used to save various pointers and register values prior to entering user subroutines.

The tables beginning at locations MNC, MNC1, and MNC2 are mnemonic op codes used by MNEDMP subroutines. Note that the second and fourth entries in the table MNC are pointers to the tables MNC1 and MNC2 respectively.

15.1.5 USING SNAP

A. RESTRICTIONS

1. Snaps may be taken anywhere in a program's instruction sequence. However, certain conditions must be avoided.
   a. Return jumps to subroutines which do not return control to L+1 can not be snapped.
   b. Instructions which are program modified cannot be snapped.
   c. If an IA is in a dump range the call to SNAPPER is dumped, not the original instructions.
2. If a user supplies an FET, the buffer defined by that FET must be at least 27 words greater than one PRU for the device which is to be written on.
3. Entry to user supplied subroutines is made only if all conditions required for a dump are met.
4. It is the user's subroutine responsibility to modify the return linkage in order to return to L+3.

B. USAGE

1. Larger buffers will tend to decrease SNAP execution times due to fewer I/O requests.
2. Writing SNACE to tape tends to decrease SNAP execution times in an active multi-programming environment.
3. The user subroutine, upon receiving control from SNAP, may modify the value of any of the twenty four operating registers by changing the appropriate SNAP provided table entry, since these tables are used to restore the registers upon exit. Furthermore,

the user may alter frequency, FWA, LWA, etc., by careful modification of the loader provided tables. The user subroutine may suppress dumps by setting bit 59 of user communication word one to one, or it may provide an alternate buffer by placing the FWA of a 13 word FET low order in the same word. The user subroutine may alter the return address from SNAP by placing a new 18 bit return address low order in user communication word two.

D. Internal Conventions

1. The following register usage is observed:

   B1 is set to one
   B6 is the FWA to be dumped
   B7 is the LWA to be dumped
   A0 is the dump increment
   X4 is the current input data word
   A7 is the current next position available in the output buffer.

2. Most of the tables and flags are order dependent and their sequence should not be changed.

E. External References

The following external references are made by SNAP. All are entry points in the routine DEBUG.

1. SNACE — The address of a thirteen word FET which defines the buffer SNACE.
2. STORE — A subroutine to store all twenty four operating registers.
3. RESTORE — A subroutine to restore all twenty four operating registers.
4. OVFLAG — The address of a location containing the overlay level number loaded during overlay processing.
5. RBO — Entry point of an eight word array of the saved index registers.

6. RAO Entry point of an eight word array of the saved address registers.

7. RXO Entry point of an eight word array of the saved X registers.

8. PCNTR A subroutine used to increment the page number used for SNAP (and TRACE) output.

15.2 TRACE

15.2.1 The TRACE routine is entered through the entry point TRACE from the routine DEBUG. TRACE consists of two portions; the trap setting routines, and the execution time evaluation routines. The latter routines evaluate triggers to determine when, if any, tracing output should be produced. All output coming from TRACE is placed on a file called SNACE.

The overall function of the trap setting routines is defined in the following steps:

a) Pick up a trace trigger
b) Process the user's program within the range specified, in conjunction with the trigger, for an instruction which matches the trigger.
c) When a match is made, set a trap in the user's word which holds the instruction. (Traps are also set at the starting and ending location for each range.)

The above steps continue until all trace triggers and all trace ranges have been processed.

A trap may be set for each instruction, hence a word has a maximum of four output producing "traps". Actually one trap is set in the user's word. The trap points to a table (ADT) entry. This table is made up of three words for each word trapped.

A trap which is set only because the word is either the start or end of a range will not cause output to be produced. These traps keep track of the frequency by which a range is processed. The range is processed once when the first instruction of the range has been processed and the last instruction of the range has been processed.

TRACE exits to either SNAP or the user, depending upon the contents of register B7.

There are two execution time tables involved with the TRACE routine:

ID Table - produced by DEBUG from the trace cards

ADT Table- produced by TRACE for each trap

When a trigger matches a user instruction a trap is placed in the word which contains that instruction. The trap is formatted as follows:

Bits	Description
59-30	A RJ to the object time subroutine which will process the data pertaining to this trap.
24	End of range flag
23	Start of range flag
17-0	Pointer to ADT table entry

The ADT table is formatted as follows:

Word					
1	User's replaced word				
2	ID1	SU	BU	BD	ED
3	ID2	ID3	ID4		

where, ED (bits 8-0) is the decrement to a word which contains a pointer to the ID table for an end of range trigger.

BD (bits 17-9) is the decrement to a word which contains a pointer to the ID table for a start of range trigger.

BU (bits 21-18) contains a flag for each section trapped, i.e., section 1 is bits 59-45, section 2 is bits 44-30, section 3 is bits 29-15, and section 4 is bits 14-0 of a word. When a bit is set (=1) that section has been trapped and may not be trapped again for that word.

SU (bits 29-22) contains binary usage flags for each section used, described as follows:

a) 00 - section has not been trapped
b) 01 - section has been trapped by a 15 bit instruction
c) 10 - section has been trapped by a 30 bit instruction
d) 11 - section is the last 15 bits of a 30 bit instruction

ID1 (bits 59-40) Bits 57-40 contain the address of the starting word of the ID table (described below) for the range which caused section 1 to be trapped. If bit 58 contains a 1, the registers are to be dumped when output is produced for this trap. If bit 59 contains a 1, the instruction which caused the trap to be set is a jump instruction.

ID2 (bits 59-40) Same as ID1, except set when section 2 has been trapped.

ID3 (bits 39-20, word 3) - Same as ID1, except set when section 3 has been trapped.

ID4 (bits 19-0, word 3) - Same as ID1, except set when section 4 has been trapped.

The ID table is formatted as follows:

Word	Bits	Description
1	59-18	The name of the range (ID on TRACE card)
	17-4	The number of words (n) of output specifications for this range.

	3	0 - 1st part of entry (entry will always be 5+n words long)
	2	1 - if registers are to be dumped
	1-0	Type of trigger specification, described as follows: a) 00 - Type TL on TRACE card - bits 17-0 contains the address referenced. b) 01 - Type TM on TRACE card; Bits 59-45 contains boolean mask Bits 44-30 contain 1st trigger mask Bits 29-15 contain 2nd trigger mask (0 if none) Bits 14-0 contain 3rd trigger mask (0 if none) c) 10 - Type TR on TRACE card; Bits 8-3 contain register name in display code Bits 2-0 contain register number in binary (0 if register is P) d) 11 - Type TM on TRACE card; Bits 59-30 contain boolean mask Bits 29-0 contain trigger mask
2	59	Start of range flag - 0 if control has not passed through the address (IA) in bits 17-0. This bit is reset to zero whenever this range is processed during overlay or segmentation modes. (Change this bit to 1 only if bit 59 of word 3 equals 1.)
	58-18	Count of number of times control has passed through the range. The count is initially zero and is reset to zero each time the range is processed during overlay

Word	Bits	Description
		or segmentation modes. (Add 1 to count each time bit 59 of word 3 is set to 1; except initially.)
	17-0	Address of start of range (IA).
3	59	End of range flag - 0 if control has not passed through this (LA) address. (Change to 1 if bit 59 of word 2 equals 1; bit is initially set to 1.)
	17-0	Address of end of range (LA).
4	59	1 if this is an overlay job - set by DEBUG.
	58	1 if this range is to be processed during segmentation mode (if it is set, process the range, reset bit 58 and set bit 57.) - set by DEBUG.
	57	Set when this range has been processed during overlay or segmentation modes. (This is used as a flag by DEBUG)
	56-51	Used during overlay mode only - 1st overlay level
	50-45	Used during overlay mode only - 2nd overlay level (These overlay levels are checked against bits 11-0 of OVFLAG; set by DEBUG.)
	44-30	The number which specifies when tracing will begin; F1 on the TRACE card. (1 if F1 is omitted)
	29-15	The number which specifies when tracing will stop; F2 on the TRACE card. If F2 is zero there is no limit. (1 if F2 is omitted)
	14-0	The number which specifies how often to trace; F3 on the TRACE card. (1 if F3 is omitted)
5	59-0	A trigger specification (described in word 1 above)
6→n	59-0	Output specifications, where n = C (bits 17-4 of word 1) +5. These specifications are defined as follows:

Type	Word	Bits	Description
OL	1	59	1 - this is a 2 word specification.
		58-47	number of specification words for this range (if this is the 1st word of all output specification words)
		5-0	number of words to be dumped
	2	17-0	starting address of dump
OR	1	59	0-this is a 1 word specification
		14-9	register name in display code (A, B, or X)
		8-6	register number (0-7)
		5-0	number of words to be dumped

n+1	59-0	Same as word 1 except bit 3=1 (2nd part of entry), except bits 17-4 are unused.
n+2	59-0	A trigger specification (described in word 1 above)

These 2 words will be repeated for each subsequent trigger specification.

15.2.2 Trap Setting Time Subroutine Descriptions

1. Initialization

The subroutine begins by saving all registers which may be needed by subsequent routine processing.

It then determines if the program being processed is in overlay or segmentation mode. In both cases only those routines just loaded will be processed (this differs from normal loading in that all generated routines aren't loaded at once).

Then a validity check is made on each range as it is processed.

The following make a range invalid.

a) No start of range specified

b) No end of range specified

c) The start of range is unsatisfied

d) The end of range is unsatisfied

e) The debugging routines are included in the range

f) No output trigger is specified

If the range is invalid it is ignored and the next range is processed.

If the range is valid it is processed by one of the following subroutines:

a) CHKA1 - processes address reference triggers

b) MASK2 - processes mask triggers

c) CHKR1 - processes register triggers

When processing for a range has been completed, subroutine SETLST is executed. This subroutine sets traps for the start and end of the range.

When all ranges have been processed the file SNACE is opened and all interface registers are restored. These registers are as follows:

a) A0 - user's field length

b) B2 - user's starting address

c) B3 - starting address of TRACE, SNAP routines

d) B4 - last address+1 of TRACE, SNAP routines

e) B5 - start of SNAP tables

f) B6 - start of TRACE tables (ID tables)

g) B7 - user's or SNAP routine's starting address

2. SETLST

The function of this subroutine is to set a pointer for the start and end address of each range. If the start or end address had already been trapped the routine performs the following:

a) fetch the address of the second word of the ADT table from the trapped word.

b) fetch the address of the next available word (in LASTENT).

c) place the difference between (a) and (b) in either BD or ED (described in 2.1 above) and

d) produce the following word:

Bits 59-42 contain address of the related ID table

e) increment LASTENT by 1

If the starting or end address had not been trapped the following is performed:

a) produce a four word entry. The first three words are the same as a normal ADT trap entry except word 3 is empty, and word 2 contains a 2 in either the BD or ED fields. The fourth word is produced as in (d) above. The trap will contain a RJ TR5.

If the start or end address had been trapped by a previous start or end trigger the following is performed:

a) Determine the address of the last 1 word (d above) entry.

b) fetch the address of the next available word from LASTENT

c) Produce the following word:

Bits 59-42 contain address of the related ID table

Bits 17-0 contain the difference between (a) and (b)

d) Update LASTENT

This subroutine is only called by the initialization.

The start of range bit is set in the trapped word, depending upon which end is being processed.

3. INSSET

   This subroutine sets the SU and BU bits in the second word of the related ADT table entry (to the user's word being processed). The bits reflect those sections of the word which have been trapped.

4. NXTWRD

   This subroutine fetches the next sequential user's word (within the specified range) and determines, by testing the BU field of the related ADT entry, its availability as a prospective trapping word. If the complete word has been trapped register X6 is set to zero. Control is returned to the calling routine.

5. SETRAP

   This subroutine builds the ADT entry and sets the trap in the user's program for words which have not already been trapped. The user's word (where the trap is to be placed) is placed in the first word. Then either ID1, ID2, ID3, or ID4 is placed in its proper field. The field positioning is determined by which section has been trapped and the contents to be placed are found in ID. A trap is set in the user's word which is formatted as described in 2.1 above. Only bits 59-30 and 17-0 are filled.

   If a trap has already been set only one action is performed; to place the contents of ID in one of the ADT table's ID fields. All jump instructions trapped have a RJ TR3 set into the user's program.

   All non-jump instructions trapped, have a RJ TR2 set into the user's program. If a trap contains mixed instructions (jump and non-jump) a RJ TR4 is set.

6. CHKA1

This subroutine processes all triggers of the TL type. If the address is unsatisfied the trigger is ignored and control is returned to the initialization subroutine. There are three possible applications of this type trigger in a word. The sections possibly usable are 1 and 2, 2 and 3, and 3 and 4. The sections are tested first for availability. If they are available the reference address is matched against the user's instruction. If the trigger matches, control is passed to a routine which will perform initialization for setting the trap.

7. MASK2

This subroutine processes TM type triggers and is divided into 2 sections; one which processes 30 bit masks and one which processes 15 bit masks.

The first section, which processes 30 bit masks, checks 3 sets of sections for availability; 1 and 2, 2 and 3, and 3 and 4. These sections are tested for availability from the left. If 2 contiguous sections are available the boolean mask is logically "and"ed to the sections. If the trigger mask matches the results of the boolean operation, control is passed to a routine which will perform initialization for setting the trap. Only 2 traps may be set in a word by this section.

The second section, which processes 15 bit masks, checks all four sections for availability. However, only the left-most 15 bits of an instruction will be checked. Each section is checked for availability, starting from the left. When an available section is found, its contents and the boolean mask are "and"ed together. If the first trigger mask matches the results, control is passed to a routine which

will perform initialization for setting a trap. This process is completed for all trigger masks present. Four traps may be set by this section.

8. WON, TOO, TRE, FOR

   These four subroutines initialize the setting of 15 bit traps into sections 1, 2, 3 and 4, respectively. These subroutines all call SETRAP and INSSET. The trap set by SETRAP will be a RJ TR2.

9. OAT, TAT, TAF

   These three subroutines initialize for the setting of 30 bit traps into sections 1 and 2, 2 and 3, and 3 and 4, respectively. All of these subroutines call CKP, SETRAP, and INSSET.

10. CHKR1

    This subroutine processes TR type triggers. If this is a jump trace (register is P) only sections 1 and 2, 2 and 3, and 3 and 4 are checked. If 2 contiguous sections are available, the instruction is checked to see if it is a 30 bit instruction. If it is, it is checked to see if it is a jump instruction. If it is, control is passed to OAT, TAT, or TAF, depending upon the position of the instruction in the word.

    If this is not a jump trace, all four sections are checked separately. If a match is found (the result register is compared) control is passed to WON, TOO, TRE, or FOR, depending upon the position of the instruction in the word.

11. CKP

    This subroutine checks for a jump instruction. If a jump instruction is found, bit 19 of ID is set to 1. If a jump is not found B4 is set equal to B5.

12. RGMTCH

This subroutine checks to see if the trigger register matches the result register of an instruction. There are four different exits to this routine, as follows:

a) 15 bit instruction, no match - X4=0, B3=0

b) 30 bit instruction, no match - X4≠0, B7=0

c) 15 bit instruction, match - X4=0, B3≠0

d) 30 bit instruction, match - X4≠0, B7≠0

15.2.3 Execution Time Subroutine Descriptions

All trap interpretation routines start by storing the user's registers.

1. TR2

The function of this subroutine is to evaluate all non-jump trapped words. The evaluation of the trapped word is performed as follows:

a) Fetch the contents of the trap

Bits 17-0 of the trap contain a pointer to the ADT entry for this word.

b) Interrogate bits 29-22 of the second word of the ADT entry for section usage.

When a trapped section is found the registers are restored, the instruction in the section is executed, the registers stored, and control is sent to OUTPUT.

Non-trapped instructions are executed in the same manner as above, except there is no output produced.

If there is a trapped jump instruction in the user's word, the word is pre-positioned to the instruction by TR4. Only the trapped

non-jump instruction is executed and control is returned to TR4.

If the start of range and/or end of range flag is set, control is passed to TR5.

2. TR3

The function of this subroutine is to evaluate all jump type trapped words. The evaluation is performed as it is in TR2.

When a trapped jump is found, an address internal to TR3 is placed in the address portion of the jump. The registers are then restored and the jump instruction is executed. If the jump does not "take", control falls through and the word is again examined. If the jump does "take", control is passed to a section in TR3. Then control is sent to OUTPUT. When control is returned, the actual jump instruction is executed. (Note: the trace will not perform correctly if a trace is attempted on a RJ which has following parameters.)

Non-trapped instructions are executed as they are in TR2. If the start of range and/or end of range flag is set, control is passed to TR5.

If the user's word contains a trapped non-jump instruction, the word is pre-positioned to the instruction by TR4. Only the jump instruction is executed and control is returned to TR4 (if control returns).

3. TR4

The function of this subroutine is to sort out jump and non-jump trapped instructions in one word. This routine executes all non-trapped instructions. If the instruction is of non-jump type, control

is passed to TR2. When the instruction is of the jump type, control is passed to TR3.

All non-trapped instructions are executed as in TR2.

If the start of range and/or end of range flag is set, control is passed to TR5.

4. TR5

When TR5 is entered the end and start trace bits (found in trap) are interrogated to determine whether this is the start or end of a range (it may be both). The following steps are performed for a start of range bit set:

a) The ADT table entry corresponding to the user's trapped word is found. Bits 17-9 of the second word of the entry contain a relative pointer to a word which contains an absolute pointer to the ID table (defined in 2.1) related to a range.

b) The start of range flag in the ID table is set and the end of range flag in the ID table reset if the following conditions are met:

   1) The number of times through the start is less than the limit
   2) The end of range flag is set (bit 59 of word 3 in ID table)

c) Process all ID tables (ranges) which have this word as a starting point by following the thread to the end. Bits 17-0 of the word which contained the previous ID pointer, contains a relative pointer to the next ID pointer. A zero in these bits indicates the end of the thread. Each pointer is processed as in (b) above.

The following steps are performed for an end of range bit (in trapped word) set:

a) Same as (a) for start except the relative pointer is found in bits 8-0.

b) The end of range flag in the ID table is set, the start of range flag in the ID table is reset, and the frequency count is incremented by 1 if the following conditions are met:

   1) The number of times (frequency count) through the end word is not greater than the limit.

   2) The start of range flag is set (bit 59 of word 2 in ID table)

c) All ID tables referring to this word as the last word of its range are processed, using the same method as in (c) above.

5. OUTPUT

The range information (in ID table), corresponding to the instruction trapped, is obtained by looking in the ADT entry (which the trap points to).

Output is produced if the following conditions are met:

a) The number of times through the range is less than the limit (F2 on TRACE card).

b) The number of times though the range meets the starting parameter (F1 on TRACE card).

c) The number of times through the range meets the frequency parameter (F3 on TRACE card).

d) Start of range flag is set.

The output appears in the following format:

TRACE PAGE xxx

OPERAND REGISTERS,
RESULT REGISTER IS

Register dump (if any)

Output dump(s)

The operand registers and their values are placed following the corresponding literal. If there is no i register, the literal "NO 1ST" will be printed. "K" represents an 18 bit address while "N" represents a 6 bit constant. The result register and its value are placed following the corresponding literal. The registers used by an instruction are found by using the RGTBl table (described in 2.4).

If a register dump is requested it is placed immediately after the result register value. The output trigger dump then is output.

A heading and page number appears at the start of each trap's output.

6. CLRLINE

This subroutine is called by OUTPUT during the dump. The function of the subroutine is to fill the dump buffer with blanks.

7. CONVERT

The function of this subroutine is to convert a binary number from binary to display code. Input to this routine is as follows:

a) X3 contains the binary number

b) B4 contains the number of digits to convert

The binary number is expected to be right justified and will be left justified upon output. The first 10 digits are placed in X6 and the remaining (if present, always 10) digits in X7.

8. OUTM

The function of this subroutine is to place output into the SNACE file's buffer. When the buffer load exceeds 100 words the buffer is dumped to SNACE. Register A2 indicates the starting address of the

output and register B4 contains the number of words to be transferred. The number must be less than 100_8. A 1 in bit position 6 indicates to the subroutine that the buffer is to be dumped.

15.2.4 Internal Table and Buffer Descriptions

1. RGTB1 - used in determining legal register usage and length of instructions. Only non-jump instructions (op-code is greater than 07) are included in the table. The format of this table is as follows:

Bits	Description
59-54	Operation Code (10_8-77_8)
53-42	Unused
41-36	j register in instruction (first register in output)
35-30	Increment in register save table (in DEBUG)
29-24	k register in instruction (second register in output)
23-18	Increment in register save table
17-12	i register in instruction (result register in output)
11-6	Increment in register save table
5-0	Length of instruction

There are two exceptions which are defined as follows:

a) When the k register = 04, the field is 18 bits in length and is defined as K in the output.

b) When the j register = 11_8, there is no j or k register. Instead there is a 6 bit register defined as N in the output.

2. MASKS - a series of eleven word length 12 bit masks starting with zero, progressing to all 7's. The mask table is used by the subroutine CONVERT to insure only pertinent data is produced.

3. REGDMP - An output buffer filled by the user's register values when the user designates a register dump. The buffer is filled from RBO, RAO, and RXO for the B, A, and X register values, respectively.
4. DMPLIN - A 120 character buffer used when dumping output, according to the user's output specification.
5. HEAD - the heading buffer which is produced each time a trap produces output. The page number is stored in HEADP.
6. TRTBL - This table is used by the initialization routine to determine which subroutine will evaluate the trap setting triggers.
7. TRTB1 - This table is used by SETRAP and SETLST to determine which subroutine will evaluate the trap at execution time.
8. NOOP - This is an execution time buffer used by TR2, TR3, TR4, and TR5. The instruction to be executed is masked into the NO-OP operations.

.2.5 Data Descriptions

1. FLUSY - Zero if entry to TR2 or TR3 was made from TR4.
2. FLUZY - Zero if entry to TR5 was made from TR2, TR3, or TR4.
3. TR19B - Original instructions at TR19. TR19 is restored when a new ID table is to be processed.
4. TR19J - A jump around code which should not be executed when 2 word ID trigger is being processed.
5. FLAGS - Used at trap time to keep track of sections already trapped in a word. Bits 15-12 are set as each section is trapped or tested.
6. LASTENT-the next available location for an ADT entry to be placed.
7. WRDTEMP-at trap time, the contents of the user's word that is being processed. At execution time it is used temporarily as a pointer by TR5.

8. TRIGR - at trap time the contents of the trap setting trigger. at execution time used as a pointer to the next ID table pointer by TR5.

9. PRESENT-At trap time the address of the second word in the presently used ADT entry. At execution time it holds the contents of the first word of the ADT entry, i.e., the user's original instructions.

10. TRAPADR-The address of the trap in the user's program.

11. PRES1 - The contents of the second word of the ADT table at execution time.

12. INSTRC -The section being processed during execution time.

13. PRES2 -The contents of the third word of the ADT table at execution time.

14. OVFLG - Non-zero when an overlay or segmentation run is in process.

15.2.6 TRACE Usage

1. Restrictions

   a) A RJ which does not return to L+1 cannot be traced.

   b) Words that are program modified cannot be traced.

2. Efficiency Hints

   a) Keep the ranges as short as possible; the initial scan processes the complete range.

   b) Do not place the start of range or end of range address in data; the TRACE routine places a trap at both locations (which may hinder operation).

   c) Assigning SNACE to tape tends to decrease overall running time in a multi-processing situation.

3. External References

a) SNACE - The name of the FET which describes the usage of the file SNACE.

b) RESTORE- The routine which replaces the user's register values into the registers.

c) STORE - The routine which places the user's register values into a storage buffer.

d) RBO - An eight word array which contains the values of the B registers, BO-B7.

e) RAO - An eight word array which contains the values of the A registers, A0-A7.

f) RXO - An eight word array which contains the values of the X registers, X0-X7.

g) OVFLAG - A table produced by DEBUG for usage during overlay and segmentation runs. During overlay runs OVFLAG, bit 59=1 and 11-0 contain the overlay level just loaded. During segmentation runs OVFLAG, bits 59 and 58=1. OVFLAG+2 contains the lowest possible location-1 which TRACE can use while building ADT entries. OVFLAG+5 contains the starting location where ADT table entries may be built.

h) PCNTR - The subroutine which computes what the next page number will be.

i) ZSQUIZH- The subroutine entry point which provides more room during segmentation for ADT table entries in the event OVFLAG+2 is reached.

4. Error Messages

Errors a, b, and c are produced by the initialization subroutine and written on SNACE. Error d is produced by OUTPUT and is also written on SNACE.

a) In the event that there is an error on the TRACE card, the following message is issued:

ERROR ON TRACE XXXXXXX CARD

where XXXXXXX is the ID name on the TRACE card.

b) In the event that the ADT tables which TRACE builds overlap with the user's program the following message is issued:

TRACE TABLES AND USERS PROG OVERLAP

This error causes the job to abort.

c) If a trace range is within the debugging routines the following message is produced:

SORRY, YOU MAY NOT DEBUG THE DEBUG ROUTINES

All of these errors cause the TRACE card data to be ignored.

d) If an output specification address in unsatisfied the output for that specification is ignored. The message produced is as follows:

ADDRESS XXXXXXX IS UNDEFINED

where XXXXXXX is the unsatisfied name.

15.3 DEBUG

DEBUG is a general-purpose program which is used for many phases of a debugging run. All of the other debugging programs link to DEBUG. It consists of several major routines which are called at various times during a debugging run by LOADER, SNAP, TRACE, or OVERLOG.

A detailed description follows each of the following routines:

1) TSCARD - Process parameters on SNAP and TRACE cards and build tables.
2) SETADR - Prepare for entry to SNAP or TRACE.
3) WRDEBUG - Write DEBUG FILE.
4) TSSEGUP - Save TRACE/SNAP tables for segmentation job.
5) ZZGETFL - Modify field length.
6) PCNTR - Page counter for TRACE and SNAP output.
7) STORE - Save registers during execution.
8) RESTORE - Restore registers during execution.

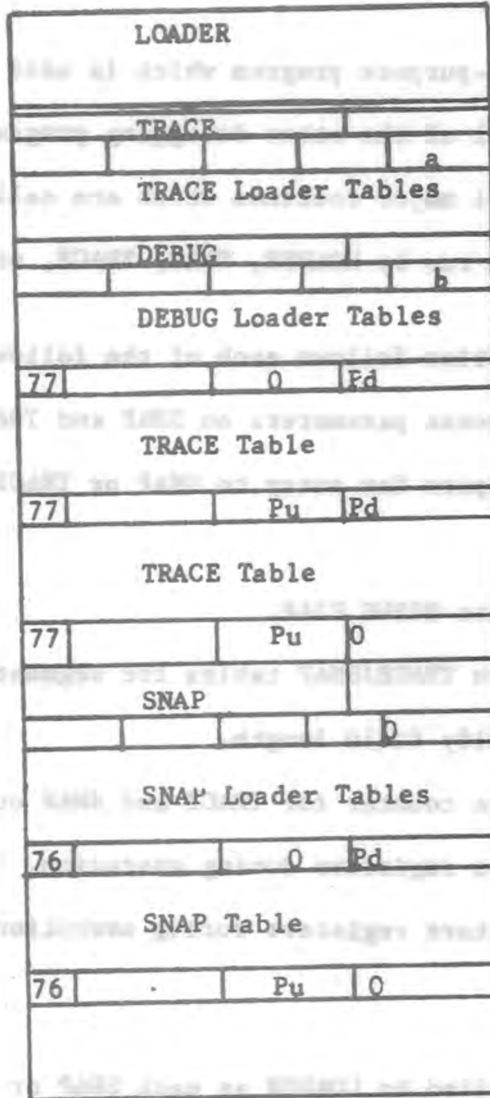
15.3.1 TSCARD

This routine is called by LOADER as each SNAP or TRACE card is processed. The parameters on the card are interpreted and SNAP or TRACE tables are built accordingly.

The following diagram shows the layout of the loader tables and the TRACE and SNAP tables after both TRACE and SNAP cards have been processed by TSCARD. In this example, there were two TRACE cards and one SNAP card.

RA+FL

FWALODR

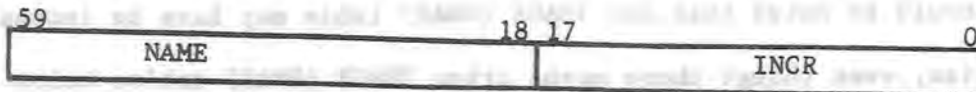


TBLNEXT

(a) and (b) are thread indices to the next loader table. (See LOADER IMS for description of loader tables.)

All TRACE and SNAP tables are enclosed with pointer words as shown in the diagram. Pd is a thread index to the next lower table, while Pu is a thread index to the next higher table. All TRACE tables must be together in one group, as shown. The same is true for SNAP tables. For this reason, all TRACE cards and all SNAP cards must be contiguous.

Reference should be made to the TRACE or SNAP description for the format of the respective table. However, there is one difference to be noted at this time: Any table words which ultimately contain an address are completed by the routine SETADR after the user's programs have been loaded. The addresses of the entry points or common names are not known until that time. TSCARD sets up these table words as follows:



If NAME refers to a COMMON block, bit 59 will be set. If the word refers to blank COMMON, then bit 59 will be set and bits 18-58 will be zero. INCR may be positive or negative.

The following sequence of events takes place in TSCARD:

1. The number of parameters on the TRACE/SNAP card is obtained from RA+64. If there are no parameters, the message "ERROR ON TRACE OR SNAP CARD" is put on the dayfile and the job aborted. At this time it should be noted that on all other parameter format errors, control goes to the location TS1. This causes the above message to be issued, but the job is not aborted. Instead, the TRACE or SNAP to which the erroneous card applies will be ignored. This is done by zeroing out the ID word of the table.
2. It is now determined whether or not this card is a continuation card. In order to be a continuation card, the following conditions must exist:
   a) The first parameter must be ID.
   b) The ID name must appear in a previous table of the same type.

   B6 is then set to the address of the pointer word (upper 6 bits = 76 or 77) at the top of the table being processed. B6 will remain this

value throughout the execution of TSCARD. If a continuation card, the parameters will be stored in fields in an already existing table. Otherwise, a new table is established by setting up pointer words so as to define limits of a minimum size TRACE or SNAP table. TBLNEXT (bits 36-53 of RA+66) is reduced accordingly.

It should be noted that any TRACE (SNAP) table may have to increase in size, even though there maybe other TRACE (SNAP) tables below it. This is because of the possibility of continuation TRACE (SNAP) cards appearing which contain parameters requiring a table to increase in size (Example: A continuation TRACE card containing any output specifications).

3. TRACE card parameters are checked for the following errors:

a) ID

1) No subparam following ID (ID name).

2) First character of subparam not alphabetic.

3) Subparam more than seven characters.

b) F1, F2, or F3

1) No subparam (frequency parameter value).

2) Subparam not numeric.

3) Subparam more than five octal digits.

c) IA, LA

1) No subparam (entry or common name).

2) First subparam not alphanumeric.

3) No second subparam if a negative increment is required.

4) Second subparam (if present) not numeric.

5) Second subparam (if present) more than 6 octal digits.

6) The same parameter has occurred before.

d) TM

1) No subparams.

2) First subparam (match key) not numeric and not equal to five or 10 octal digits.

3) Succeeding subparams (masks) not numeric.

4) Succeeding subparams more than five digits if first subparam is 5 digits.

e) TR

1) No subparam (register).

2) Subparam not a register designator (AO-A7, BO-B7, XO-X7, P).

f) TL

1) No subparam (entry or common name).

2) First subparam not alphanumeric (unless TLB).

3) No second subparam if a negative increment is required.

4) Second subparam (or first if TLB) not numeric or more than 6 octal digits.

g) OR

1) Two subparams not present.

2) First subparam is not a register designator (AO-A7, BO-B7, XO-X7).

3) Second subparam is not numeric, or is more than two octal digits.

h) OL

1) At least two subparams not present.

2) First subparam not alphanumeric (unless OLB)

3) Second subparam (first of OLB) not a numeric increment of six or less octal digits if an increment is required).

4) Third subparam (word count) not present if an increment (second subparam) was required.

5) Word count subparam not numeric or has more than two octal digits.

4. SNAP card parameters are checked for the following errors:

a) ID, F1, F2, and F3 are checked in the same manner as the corresponding TRACE parameters.

b) INT

1) No subparam (dump interval).

2) Subparam not numeric or consists of more than four octal digits.

3) Subparam has a value of zero.

c) F

1) No subparam (dump format).

2) Subparam not alphanumeric or consists of more than two characters.

d) IA, FWA, LWA

1) IAB

2) FWAB1 or LWAB1

3) No subparams

4) Any alphanumeric subparams more than 7 characters.

5) No negative increment subparam if one is required. Must be numeric and six or less digits.

6) The same parameter has occurred before.

e) UR

1) No subparam (user entry point name).

2) The UR parameter has occurred previously.

5. After all of the parameters have been processed on a SNAP or TRACE card, TSCARD exits to LOADER. LOADER then drops the central processor in order that the SCOPE operating system will advance the job to the next control card.

If any of the format errors described above occur, the ID word of the appropriate table is cleared, and this same exit is taken immediately.

6. The following subroutines are used by TSCARD:

   a) GETPARAM - The next parameter is picked up from the parameter area beginning at RA+2. If, on entry, B0=0, and no parameters remain, the error exit (TS1) is taken.

   b) CKA - The parameter in X1 is checked to see if it is alphanumeric and if it is not more than seven characters long. If these conditions are not met, the error exit (TS1) is taken.

   c) CKN - The parameter in X1 is converted from display code to octal. If any non-octal digits are encountered and if B1=0, the error exit (TS1) is taken.

   d) LOWER - If a continuation card makes it necessary to increase the size of the table being processed, this routine is called in order to move down all of the tables below the one being processed. The routines TBLLOWER and UPPS are called by this routine.

   e) TBLLOWER - This routine is called to decrement TBLNEXT (bits 36-53 of RA+66) whenever it is necessary to expand the TRACE or SNAP tables downward in core. If TBLNEXT and CORNEXT meets, there is no more available core. The message "TRACE OR SNAP TABLES OUT OF CORE" is issued to the dayfile and the job is aborted.

   f) UPPS - Whenever a table has to be enlarged, this routine is called in order to increase the thread indices (Pu and Pd) so as to correctly reflect the new length of the table.

8) MOVEDOWN - This routine is called whenever a TRACE output specification parameter is processed. All additional trigger specifications (if any) are moved down. This is necessary because both the number of words of output specifications and the number of words of additional trigger specifications are variable, and the trigger specifications reside at the end of the table. Before this routine is called, the routine LOWER is called to accommodate the enlargement of this table.

15.3.2 SETADR

This routine performs all final initialization just prior to entering SNAP or TRACE. It is called by LOADER at the point in time when the user's entry would be taken if this were not a TRACE or SNAP run. It is also called by LOADER after a segment user call has been processed. On overlay runs, it is called by OVERLOG prior to returning to the user.

SETADR does the following:

1) The user entry address, which is passed to SETADR in B7, is saved.
2) The address of the top of the loader tables is established. For normal and segment runs, the address is found in RA+67 (FWALODR). For overlay runs, it is found in OVFLAG+4. (See section 3.10.)
3) The TRACE and SNAP flags in RA+67 are checked to determine whether SNAP, TRACE, or both are requested. The routine FINDP is called to fetch the entry address of SNAP and/or TRACE.
4) The loader tables are searched for the TRACE/SNAP tables. When the TRACE tables are found, TRACEF is called to search the loader tables

for the entry point and common block names in the TRACE tables. The addresses are set in the corresponding words in the TRACE tables. SNAPF does the same processing for SNAP tables.

5) The addresses of the upper end of the TRACE and SNAP tables are saved in locations TTRACE and TSNAP, respectively. These will be passed to SNAP and TRACE.

6) A scan of the loader tables is made to determine the upper and lower limits of the core used by the routines TRACE, SNAP, and DEBUG. The upper limit is stored at TSUP, and the lower limit, at TSDOWN. These values are also passed to SNAP and TRACE.

7) The special dump bits in RA+66 are checked, and if set, WRDEBUG is called to write the DEBUG file so that DMP may produce a labeled and change dump. If this is an overlay run, WRDEBUG is not called, because it was already called from OVERLOG.

8) If this is an overlay run, unneeded field length is now released by calling ZZGETFL for a field length equal to that of the original + FLINCR (see OVERLOG description).

9) The index registers are now set up as follows for the entry to TRACE or SNAP:

   a. B7 = The address to which TRACE will pass control. This is either the user entry, or the entry address of SNAP if both TRACE and SNAP are being used.

   b. B6 = The address of the top of the TRACE tables.

   c. B5 = The address of the top of the SNAP tables.

   d. B4 = The LWA+1 of the core occupied by the debugging routines.

   e. B3 = The FWA of the core occupied by the debugging routines.

f. B2 = The address to which SNAP will pass control. This is always the user entry.

g. A0 The field length.

The following subroutines are used by SETADR:

a. SNAPF - This routine is called to complete the processing on each SNAP table. OVSEGTBL is first called to determine whether or not this table is to be processed by SNAP at this time. If, upon return from OVSEGTBL, B3=0, SNAPF exits without processing the table. Otherwise, for each table word with an address to be set, the routine TRACEFF is called with B4= the address of the word to process.

b. TRACEF - This routine performs the same process on TRACE tables as performed on SNAP tables by SNAPF.

c. OVSEGTBL - This routine determines whether a TRACE or SNAP table is to be processed at this time. B3 is set to zero if the table is not to be processed. The following steps are taken.

1. B3 is set to zero if the ID word is zero.
2. If a normal run, the table may be processed.
3. Since it is an overlay or segment run, bit 59 is now set in the overlay/segment control word.
4. If this is a segment run, step 8 is taken.
5. The overlay level is fetched from the overlay/segment control word (word 4 of table). If the level is not 0,1 (the preset value), then the table has already been processed. Since tables

pertaining to overlays are processed only once, exit is made with B3=0.

6. A call is made to either FINDP or FINDC (whichever is appropriate) to attempt to locate the address for the IA field in the table. If the address is not found, then the overlay pertaining to this table is not yet loaded, and hence this table is not ready to be processed. If IA is an absolute address, then this step is skipped.
7. A check is made to see if the above address lies within the limits of the last overlay load. If so, it is time to process this table. An exit is made with B3 non-zero.
8. Tables pertaining to segment runs may be processed more than once. As a result, it is necessary to keep two copies of each table in core for segment runs. The upper copy is never changed from its original form as set by TSCARD. Steps (6) and (7) are taken using the upper copy of the table. If it is determined that the table should be processed, the upper copy is moved down over the lower copy, thus restoring the names of entry points and common blocks.

d. TRACEFF -This routine is called by TRACEF and SNAPF with B4=the address of name whose address is to be found. If the name is a common block, FINDC is called. If the word refers to blank common, then the address of blank common is obtained from FWALODR-2. Otherwise, FINDP is called.

e. FINDC - This routine searches the loader tables for a labelled common block name that matches the name in X1. If found, the address is returned in X1. Otherwise, X1 is returned =0.

f. FINDP - This routine searches the loader tables for an entry point that matches the name in X1. The address is returned in X1, if a match is found. Otherwise, X1 is returned -0.

15.3.3 WRDEBUG

This is the routine which does the writing of the DEBUG file. Depending on the circumstances, it may be called from LOADER, OVERLOG, or SETADR (in DEBUG). The following events take place:

1. A CIO call is issued to rewind the file DEBUG.
2. The upper and lower limits of the LOADER tables are determined. If this is an overlay run, these limits are obtained from the OVFLAG area (see section 3.10); otherwise, FWALODR (bits 0-17 of RA+67) will be the upper limit and TBLNEXT (bits 36-53 of RA+66) will be the lower limit.
3. The loader tables are scanned for common block declarations within the tables for each program. Within each program, such references to common blocks are sorted so that the lowest addresses will appear first. This does not affect any loader processing, and it simplifies the processing of common names for DMP.
4. A call is made to the routine TURN. This routine completely reverses the order of the loader tables. This is done so that it will be more convenient for DMP to read the loader tables. With the upper end of the tables at the beginning of the DEBUG file, DMP will be able to pick up the first thread index, even if all of the tables are not

read on the first read.

5. The CIO parameters are set up as follows and the loader tables are written to the DEBUG file:

   FIRST = TBLNEXT

   IN = FWALODR+1

   OUT = TBLNEXT

   LIMIT = FWALODR+2

6. A second call is made to TURN so as to restore the loader tables to their proper order.

7. If no change dump is requested, the exit to the calling program is now taken.

8. The field length is increased by 100_8 words. This is done because it is necessary to write the entire original field length to the DEBUG file, and the FET may not be within the circular buffer.

9. The CIO buffer parameters are set up as follows, and, as a result, the entire (original) field length is written to the DEBUG file. The FET is origined at the original field length (A0) +2.

   FIRST = 0

   IN = OLD field length

   OUT = 0

   LIMIT = Old field length +1

10. The field length is reduced to its original value by calling ZZGETFL.

11. WRDEBUG exits to the calling program (LOADER, OVERLOG, or SETADR).

15.3.4 TSSEGUP

This routine is called by LOADER for segmentation jobs which use TRACE and/or SNAP. Since it is called before the loading of SEGZERO, the TRACE/SNAP tables are still in core. Also, it is the copy of DEBUG that

was in core when the TRACE/SNAP cards were processed in which TSSEGUP is executed. As a result, TSSEGUP does not set any pointers in the OVFLAG area, since DEBUG will be reloaded as part of SEGZERO.

The purpose of TSSEGUP is to move the TRACE/SNAP tables so they will be preserved as segments are loaded. This is done by increasing the field length by the amount defined by the symbol UPFL (2000_8 in the release version of DEBUG). The TRACE/SNAP tables are then found and are moved up in core. The core allocation of the newly acquired field length will appear as follows:

Address	Contents
RA+FL+UPFL	
	SNAP tables (copy 2)
RA+FL+3+2a+b	
RA+FL+3+a+b	TRACE tables (copy 2)
RA+FL+3+a	SNAP tables (copy 1)
RA+FL+3	TRACE tables (copy 1)
	b
	a
Original RA+FL	Out of Bounds Jump
RA	

The out-of-bounds jump is placed at the original RA+FL because the loader fills references to unsatisfied externals with this address. A jump to this location will still result in an out-of-bounds jump.

(a) equals the length of one copy of the first type of table (TRACE in this case).

(b) equals the length of one copy of the second type of table. (b) = 0 if there is only one type. Either the TRACE or the SNAP tables may appear first, depending on the order of the TRACE and SNAP control cards.

The reason two copies of each table are needed is explained in the description of OVSEGTBL.

15.3.5 ZZGETFL

This routine is called in order that the field length be changed. On entry, X6 contains the desired field length. If the value in X6 is equal to the current field length, no call to MEM is made.

15.3.6 PCNTR

This routine keeps track of a page number in display code which is preset to zero. When the routine is entered the page number is incremented and returned in X6. In this way intermixed TRACE and SNAP outputs both use the same counter for numbering pages.

.3.7 STORE

This routine is used by TRACE and SNAP to save the user's registers. All 23 registers are saved. The process involves first saving B7 by testing each of the 18 bits of B7 in the sign position, and executing a RJ * for each bit not set. When this process is completed, an array will exist which will contain one's in the low-order bit position of each word in which the RJ was not performed. Thus B7 has been saved without changing the contents of any other registers.

The remainder of the registers are saved without any difficulty, the contents of B7 are then determined, and the RJ array is reinitialized for the next time through STORE.

15.3.8 RESTORE

This routine is used by TRACE and SNAP to restore the user's registers prior to returning to the user. All 23 registers are restored. The technique is as follows:

1. All registers are restored except A1, A0, X0, and B1-B7.
2. A0 is set to -0.
3. X1 (restored at this time) is unpacked into B1-B6 with the sign of X1 in B7.
4. X0 and A1 are restored.
5. X1 is repacked from the registers in (3) above.
6. B1-B7 and, last of all, A0 are restored by passing through an array which was set up in STORE. The array is of the following format:

```
SB1    A0+K
 .
 .
 .
SA0    A0+K
```

K is the saved value of the respective register. Note that A0 was set to -0 in step 2 so that values of -0 would be correctly restored. Also, note that A0 is the very last register restored.

15.3.9 Flags, Constants, Temporaries, and Buffers

Bits used in loader communication area:

RA+66 (DEBUG control bits)

Bit 28=1 if labelled dump requested.

Bit 29=1 if change dump requested.

Bit 30=1 if TRACE is to be used with overlay or segment job.

Bit 31=1 if SNAP is to be used with overlay or segment job.

RA+67 (SNAP/TRACE control bits)

Bits 32-33 are non-zero if TRACE run.

Bits 34-35 are non-zero if SNAP run.

OVFLAG - This is the start of a 6-word array used for overlay and segment runs. OVFLAG is an entry point to DEBUG, and it is referenced by SNAP, TRACE, OVERLOG, and LOADER. For overlay runs, the array has the following format:

OVFLAG:	Bit 59=1, bits 0-11 = overlay level last loaded.
+1:	Original field length.
+2:	LWA of TRACE and/or SNAP tables.
+3:	LWA of SNAP and/or TRACE tables. (=0 if TRACE and SNAP are not both being used.)
+4	LWA of loader tables for overlay last loaded.
+5	Unused.

For segment runs, the array has the following format:

OVFLAG:	Bit 58=1, bit 59=1.
+1:	Original field length.
+2:	LWA of all TRACE and SNAP tables.
+3:	Unused.
+4:	Unused.
+5:	Current pointer for TRACE ADT tables. (On non-segment jobs, TBLNEXT is used for this pointer.)

SEGF - This is a first-time flag used by SETADR in performing initialization of OVFLAG+1, OVFLAG+5, and FL during segmentation runs. It is preset to one.

PAGE - Current page number in display code.

WDFET - 5-word area used by WRDEBUG for FET when writing loader tables to DEBUG.

UPFL - Amount the field length is increased for segmentation jobs using SNAP or TRACE.

ZZFLINCR - The amount the field length is increased for overlay jobs is stored here so SETADR may have access to it. The value is defined by the symbol FLINCR in OVERLOG.

FL - Current field length kept by ZZGETFL.

RBO - Start of register storage area used by STORE and RESTORE. Locations RBO, RAO, and RXO in this area are entry points for the convenience of SNAP and TRACE.

TSPARAM - Number of parameters processed by TSCARD.

TSP - Pointer to start of table currently being built by TSCARD. Used only until this value is stored in B6.

TSZ - Number of words in tables lower than the table currently being built by TSCARD. Used by LOWER to determine how many words to move.

TSCONT - Non-zero if TSCARD is processing a continuation TRACE or SNAP card.

TSPL - Last named entry name on SNAP card currently being processed by TSCARD. Implied entry names for FWA and LWA parameters are derived from this location.

Parts of the code in DEBUG are dependent upon the order of the following locations described (TSUSER-HDEBUG). Therefore, their order should not be changed.

TSUSER - User entry address. SETADR exits with this in B2 (and in B7 if TRACE and SNAP are not both used).

FTRACE - Set to 1 by SETADR if this is a TRACE run.

FSNAP	- Set to 1 by SETADR if this is a SNAP run.
FTS	- Used to temporarily hold the sum of FTRACE and FSNAP.
ETRACE	- Entry address of TRACE.
ESNAP	- Entry address of SNAP. SETADR exits with this in B7 if this is both a TRACE and SNAP run.
TTRACE	- Address of top of TRACE tables. SETADR exits with this in B6.
TSNAP	- Address of top of SNAP tables. SETADR exits with this in B5.
TSUP	- LWA+1 of debugging routines. SETADR exits with this in B4.
TSDOWN	- FWA of debugging routines. SETADR exits with this in B3.
HTRACE	- TRACE in left-justified display code.
HSNAP	- SNAP in left-justified display code.
HDEBUG	- DEBUG in left-justified display code.
SNACE	- FET and buffer for TRACE and SNAP output. SNACE is an entry point and is the first word of a 13-word FET for the file SNACE. The buffer immediately follows the FET.

15.4 OVERLOG

When the debugging aids are to be used with overlay jobs, OVERLOG replaces OVERLOD in the 0,0 overlay. The following is done by OVERLOG:

1. After the 0,0 overlay has been loaded, an immediate entry to OVERLOG is made. During non-debug runs, OVERLOD does not get control at this time. OVERLOG gets control because it has a transfer symbol on its END card.

   Upon this initial entry, a search is made for TRACE or SNAP tables. It is required that any TRACE or SNAP cards must appear immediately before the control card which begins loading the overlay file. Thus, the tables for any TRACE/SNAP cards will still be in core. As soon as any TRACE or SNAP tables are found, the field length is increased. This is done by calling the routine ZZGETFL, which is in the program DEBUG. The amount the field length is increased is governed by the symbol FLINCR. The TRACE/SNAP tables are then moved to an area just above the original field length, in order that they will not be lost when other overlays are loaded.

   Processing continues at step 3.
2. When OVERLOG is entered to process user calls to load additional overlays, a routine called OVERLOD (within OVERLOG) is called to perform the usual OVERLOD function of calling LDR to load the requested overlay.
3. A check is now made to see if SNAP, TRACE, or a labeled dump has been requested. If so, the loader tables are read from the overlay file, and are placed above the TRACE/SNAP tables, if any.
4. If a labelled dump was requested, a call is made to the routine WRDEBUG (in DEBUG program) to write the loader tables to the file DEBUG.
5. If TRACE or SNAP was not requested, OVERLOG now exits to the user.

Otherwise, FWALODR (bits 0-17 of RA+67) is now changed to indicate that the loader tables are at a higher location in core, and the final exit is made to SETADR (in DEBUG program). SETADR will complete the processing of the TRACE/SNAP tables and will call TRACE or SNAP.

The following symbols and locations are of interest:

USERADR - This is at OVERLOG-1. The user's return address is set here after OVERLOD is called. Also LOADER stores the initial user entry in this location.

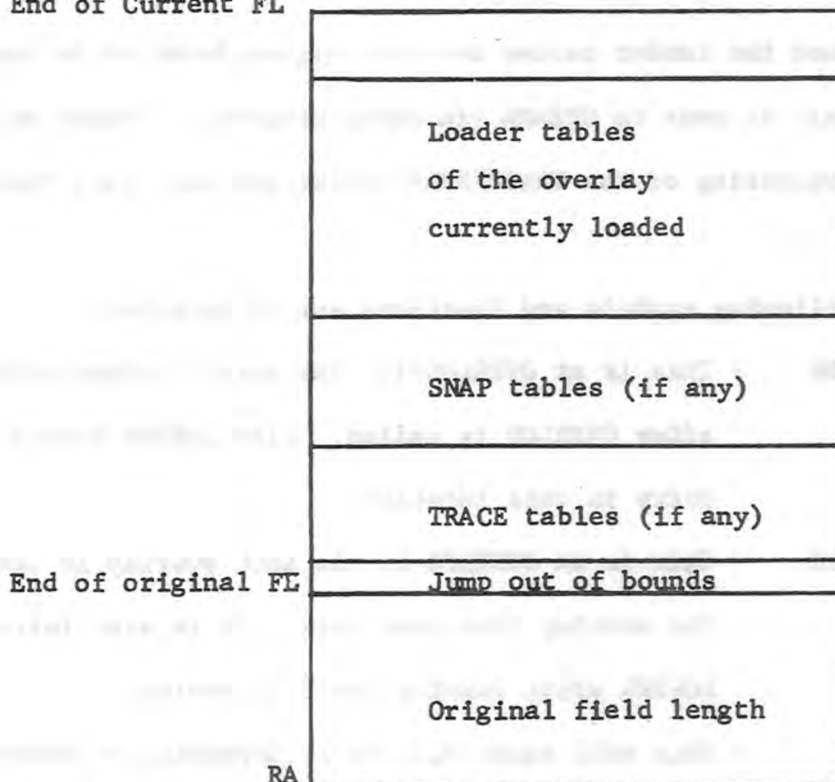
OVLFILE - This is at OVERLOG-2. As each overlay is loaded, OVERLOG stores the overlay file name here. It is also initially set by LOADER after loading the 0,0 overlay.

N - This will equal 0,1, or 2, depending on whether the user has requested no TRACE or SNAP, TRACE alone or SNAP alone, or both TRACE and SNAP, respectively.

FL - The current value of the field length is stored here.

FLINCR - This is an EQU which defines the amount the field length is increased whenever an increase has to be made. The initial release of OVERLOG has FLINCR = 400_8.

The following shows the format of high core:



The jump out of bounds is inserted at the top of the original field length because the LOADER fills unsatisfied externals with out-of-bounds jumps which are jumps to this location.

If there are no TRACE or SNAP tables (N=0), the loader tables will be read immediately above the out-of-bounds jump. If both SNAP and TRACE tables are present, their order will depend on the order in which the TRACE and SNAP cards appeared, and is immaterial.

When control is returned to the user, the field length will be equal to the original field length + FLINCR. It will be set back to this value by SETADR if a larger amount was needed to hold the loader tables.

15.5 ERROR MESSAGES

The following is a list of all error messages issued by the debugging routines, the routine from which the message appears, and a reference to the description of the message:

1) SNAP CARD PARAMETER ERROR, ID = XXXXXXX

   (SNAP, section 15.1.1)

2) ERROR ON TRACE XXXXXXX CARD

   (TRACE, section 15.2.6)

3) TRACE TABLES AND USERS PROG OVERLAP

   (TRACE, section 15.2.6)

4) SORRY, YOU MAY NOT DEBUG THE DEBUG ROUTINES

   (TRACE, section 15.2.6)

5) ADDRESS XXXXXXX IS UNDEFINED

   (TRACE, section 15.2.6)

6) ERROR ON TRACE OR SNAP CARD

   (DEBUG, section 15.3.1)

7) TRACE OR SNAP TABLES OUT OF CORE

   (DEBUG, section 15.3.1)

8) DEBUG CARD OUT OF ORDER

   (2TS)

9) TRACE OR SNAP CARDS NOT CONTIGUOUS

   (LOD)

15.3 ERROR MESSAGES

The following is a list of all error messages issued by the debugging routines, the routine from which the message appears, and a reference to the description of the message:

1) SNAP CARD PARAMETER ERROR, ID = XXXXXXX

   (SNAP, section 15.1.1)

2) ERROR ON TRACE XXXXXXX CARD

   (TRACE, section 15.2.6)

3) TRACE TABLES AND USERS PROG OVERLAP

   (TRACE, section 15.2.6)

4) SORRY, YOU MAY NOT DEBUG THE DEBUG ROUTINES

   (TRACE, section 15.2.6)

5) ADDRESS XXXXXX IS UNDEFINED

   (TRACE, section 15.2.6)

6) ERROR ON TRACE OR SNAP CARD

   (DEBUG, section 15.3.1)

7) TRACE OR SNAP TABLES OUT OF CORE

   (DEBUG, section 15.3.1)

8) DEBUG CARD OUT OF ORDER

   (2TS)

9) TRACE OR SNAP CARDS NOT CONTIGUOUS

   (LOD)

16.0 UNIT RECORD I/O

16.1 Janus I/O Package

16.1.1 1IQ General

1IQ is a PP program, always loaded at 1000B. It is called when "n.JANUS." or "AUTO." is typed at the console, to set up a control point that reads jobs through from one to four card readers, prints output files through from one to twelve printers, punches output files through from one to three card punches, and may call user-provided programs 1PL and 1FM to make plot and film units (up to four in all) handle output files.

Card readers, printers, and punches are driven by another PP program called 1IR, which is called by 1IQ. A third program, 1IS, is also called by 1IQ when it is setting up the control point; 1IS merely copies a "library" of overlays for 1IR into the field length. Thereafter, 1IS drops itself and is never called again.

After everything else has been set up, and 1IS has done its work, 1IQ calls 1IR for the first time, requests itself (1IQ) with a 2-second delay, and drops itself. Thereafter, 1IQ recalls itself in this way about every 2 seconds. Each time it comes up, it does four things: (1) Calls 1IR if it is not active at the control point. (2) Picks up type-ins which DSD may have left in the control point area, translates them, and distributes them to the FET's of the appropriate files, where 1IR (and 1PL and 1FM if they exist) can find them conveniently. (3) Expands or contracts the field length if desirable and possible. (4) Calls 1PL and/or 1FM if necessary.

Entry Information

The input register initially contains

```
VFD     18/0H1IQ,3/0,3/n,36/0
```

where n is the control point number. Called by this, 1IQ requests field length and drops itself after requesting its own return with two seconds delay. If the field length is zero, 1IQ requests a minimum 100B words and makes the future request to itself identical with the above. Otherwise, 1IQ calculates the full initial storage requirement for JANUS and requests it, and provides that the next 1IQ request will be:

```
VFD    18/0H1IQ,3/0,3/n,24/0,12/f
```

where f is the required field length divided by 100B. Until this field length is found to have been granted, 1IQ keeps requesting storage and recalling itself in this way. Once the field length is correct, all later calls to 1IQ have the form:

```
VFD     18/0H1IQ,3/0,3/n,6/a,6/a+b,6/a+b+c,6/a+b+c+d,12/f
```

where a is the number of card readers in the EST but not more than 4, b is the number of card punches but not more than 3, c is the number of line printers but not more than 12, and d is the number of film, plot, and hard-copy units but not more than 4.

Whenever 1IQ alters the field length, it changes the value of f in the call to itself accordingly.

Exit Information

The only program to which 1IQ passes information at the moment of terminating itself is itself, via its delayed PP call. The different forms of the call and the subsequent input register contents are given above.

Other Programs Called

1IS is called to another PP twice, during initialization: (1) with a call of VFD 18/0H1IS,3/0,3/n,36/0 where n is the control point number. This call is made the first time 1IQ recalls itself and finds that the field length is non-zero. 1IS calculates the total length of the overlays that it will later store in central memory within the control point's field length, and leaves it in bits 0-11 of RA+17B. This enables 1IQ to calculate the initial JANUS storage requirement.
(2) when initial storage has been obtained, 1IQ calls 1IS with a call:

```
VFD     18/0H1IS,3/0,3/n,24/0,12/v
```

where n is the control point number and v is 16(a+b+c+d+1); a, b, c, and d being equipment counts as above under "Entry Information". V is so defined because 20B words (16) are left unused for the moment at the beginning of the field length; then 20B words per equipment are set aside for an FET; beginning at the next word, RA+v, 1IS will store the overlays (for 1IR) which it copies out of itself.

1IR is called to another PP just before 1IQ drops itself, except during initialization, and unless 1IR is known to be already active. The call has the form:

```
VFD     18/0H1IR,3/0,3/n,6/a,6/a+b,6/a+b+c,6/a+b+c+d,12/w
```

where n is the control point number, a, b, c, and d are equipment counts as above, and w=v+q+200B*a, rounded up if necessary to a multiple of 100B. As explained above, RA+v is the address at which 1IS begins to store overlays; q is the total length of these overlays, stored by 1IS in bits 0-11 of RA+17B. Beginning at RA+v+q, one area of 200B words is allowed for each card reader to be driven by JANUS, and after the

end of the last such area, the file buffers of BUFLG words each begin. Thus w is the relative address of buffer number 0. The storage that 1IQ initially requests is w+3*BUFLG words, allowing three buffers to start with. (This is expanded when necessary to provide more buffers, if possible; and is contracted when 1IQ finds that one or more buffers at the end of the series are unused; but contraction does not go beyond the point that would leave one buffer free. For each buffer, 1IR will check about every thirty seconds to see whether it is the highest-address buffer in use, and if so will free it and begin using a lower-address buffer if any is available.) The 200B-word area allowed for each card reader is used by 1IR as a buffer to accumulate error messages concerning input files read through that reader. If such messages are generated during the reading of a job, they are written out by 1IR on a file that will become the OUTPUT file for the job, when it eventually begins to be executed.

Narrative

At BRW, 1IQ begins by setting constants 1, 3, and 5; reading its input register, setting D.CPAD, filling the control point number into the future calls to 1IS, 1IR, 1PL and 1FM, setting the control point job name word to JANUS, and reading the control point status word. The error flag may be zero, F.ERPP, or something else. If neither zero nor F.ERPP, it has been set by an "n.DROP." or "n.KILL." type-in. 1IQ should not initiate new file activity or request more memory, so we zero the switch instructions at BRWD and KAL. But otherwise 1IQ has to proceed normally to handle type-ins and displays for the files that are to be completed before JANUS drops; so we go to BRWA to begin. If the error flag is F.ERPP, this means that JANUS was in the condition just described, but 1IR has completed all the files and altered the error flag to F.ERPP before dropping itself; hence 1IQ may now drop itself, thus dropping JANUS. If the error flag is zero, we go straight to BRWA. At BRWA, D.RA and D.FL are set. If the field length is 0, initialization is in the very first stage, and we go to RQT. To find the total initial storage requirement, call 1IS to find the size of the overlay library; but 1IS will store this at RA+17B; so 100 words of field length are requested at RQT; after which, at PPREC, 1IQ recalls itself with a 2-second delay, using its input register to supply the future call, and then drops. When 1IQ next comes up, if the field length is not 0, it is compared with bits 0-11 of the input register; if this byte is different, go to RQS; if the byte is 0 we are at the stage when 100B words of FL have just been obtained.

At RQS, if bits 0-11 of the input register are non-zero, they are the required field length/100B, which we have not yet obtained, and we go to RQSM to request it again. But if they are zero we have just obtained 100B words of FL, and call 1IS for the first time. Subroutine CALL1IS calls 1IS and waits until 1IS has dropped itself. As bits 0-11 of the call to 1IS are 0 this time, 1IS will not store the overlay library in the field length, but merely calculate its length, and store the library length at RA+17B. Field length is going to be allocated thus: the first 20B words for utilities;

then 20B words for an FET for each device to be driven by JANUS; then the overlays; then from 0 to 77B words of waste, to make the next allocation begin at an even multiple of 100B; then 200B words for each card reader to be driven by JANUS, to accumulate messages; and finally 3*BUFLG words to provide three buffers. The number of buffers may be varied automatically between 1 and the number of devices being driven; but the allocation below the beginning of the first buffer (number 0) remains unchanged. So now subroutine CDEV is called to count the numbers of relevant devices in the EST; then the length of the overlay library is fetched from RA+17B; then the full initial storage request is set up; the wanted field length/100B is copied to bits 0-11 of the input register; the storage request is issued, and control goes to PPREC to set up a recall and drop the PP.

1IQ comes to BRWB when it finds, on being called, that it has the correct field length. Now, if bits 12-23 of the input register = 0, this is the first time 1IQ has been so called, and more initialization must be done; otherwise branch to BRWC to begin the ever-repeated part of the program. If it does not go to BRWC, it calls subroutine CDEV to count the relevant devices; then puts the number of card readers in bits 30-35 of the input register, the number of card readers plus card punches in bits 24-29, the number of readers plus punches plus printers in bits 18-23, and the number of readers plus punches plus printers plus film and plot units in bits 12-17. Call these numbers p, q, r, and s. Then s is the total number of devices JANUS may be handling, and s FET's, numbered 1 through s, are needed. Card readers will use FET's 1 through p; punches will use FET's p+1 through q; printers will use FET's q+1 through r, and film and plot units will use FET's r+1 through s. The starting address of any FET is found by multiplying its number by 20B, i.e. LDD D.RA; SHN 2; ADD PS; SHN 4 (where PS contains the FET number) - a frequently occuring sequence, gives the starting address of the FET currently being treated. Just after BRWC, bits 12-35 of the input register are copied into the same position of the call to 1IR, so that it will have available the range of FET numbers for each type of equipment. Next control point locations 60B through 150B are zeroed; then bits 48-59 of control point +111B are set = 1. (See below under subroutines FETCH and PUT. That byte must only have values 1, referring to 1IQ, and 3, referring to 1IR. If left 0, the interlock scheme described in those subroutines would lock out both programs forever.) Next find the last +1 address of the last FET = 20B*(s+1), as s is defined above. This is the beginning of the area in which 1IS will store overlays, so it is stored in bits 0-11 of the call to that program. Then (at BRWBB) zero out all the FET areas, and call 1IS.

Now a word is set up at control point +77B, which will be constantly used by 1IQ and 1IR in assigning and releasing buffers. Bits 48-59 are set to the current value of the seconds clock, +15. At KC, further down in 1IQ, the current value of the clock is compared with this number, and if the time in CP+77B has been reached, 1IQ attempts to

reduce field length. In other words, this attempt is made about every 15 seconds. Bits 36-47 are set to 3, which is the number of buffers, free or assigned, now in the field length. Bits 24-35 are set to 0, the highest number of buffers that have been in use at any one time since the last (of course at this moment there really was no last) time 1IQ tried to reduce field length. Bits 12-23 are set to 0, the number of buffers currently assigned. Bits 0-11 are left 0, and they will always remain so. The five words at CP+100B through CP+104B are considered as 25 bytes, of which 1-23 will record assignment of buffers 1-23 (23 is the maximum number of devices JANUS can handle) and bytes 24-25 are never used. These words are already 0, indicating that no buffers are assigned at the moment.

As these registers and counters in CP+77B through 104B are used by both 1IQ and 1IR, it is necessary to provide an interlock. Both programs have a subroutine called FETCH, which satisfies the interlock and copies CP+77B into five bytes beginning at KEY, and CP+100B through 104B into 25 bytes beginning at BUFAS. Only after FETCH is called may they be altered. Subroutine PUT must be called, whether or not they have been altered, to update KEY+2 (the maximum number of buffers recently in use) if KEY+3 now contains a higher number; to replace them in the control point area, and to release the interlock.

The final point of the initialization is to set bit 12 of CP+153B to 1, which identifies the control point as an "OUTPUT" one (this is also done by 1OT in setting up ordinary "OUTPUT"), and to copy the input register into CP+75B, whose only function is to provide a further identification of the control point as JANUS.

Now control reaches BRWC to begin the non-initialization part of 1IQ. First the address of the first buffer, number 0, is found. Reading RA+17B into D.Z1 through D.Z5 gives the length of the overlay library in D.Z5. Bits 12-17 of the input register give the total number of FET's; 1 is added to this representing the first 20B words of field length, preceding the first FET; the sum is multiplied by 20B to give the end +1 address of the last FET; the overlay length is added; the total is rounded up if necessary to a multiple of 100B, and the result of dividing it by 100B is saved in D.Z1. From bits 30-35 of the input register is obtained the number of card reader FET's; double it, add it to D.Z1, and multiply the result by 100B. This gives the starting address of buffer 0, which is inserted in bits 0-11 of the call to 1IR. Bits 12-35 of the 1IQ input register are, in the process, copied into the same position of the 1IR call. The start of buffer 0 is calculated as above because the overlay library follows the last FET; then 0 to 63 CM cells are wasted, up to the next address divisible by 100B; then 200B words are allowed for each card reader FET to accumulate error messages generated while reading a job through the card reader; and immediately after the last of these begins the first buffer.

Now call subroutines CONSOLE and STATUS. They are called here, just as 1IQ begins its normal work, and they will be called again just before it drops itself. CONSOLE takes operator type-ins that DSD may have put into CP+60B, translates them, and passes them to the relevant FET's. STATUS looks at words 10B through 15B of each active FET, and if they appear to contain a display message, copies them into one of the 6-word areas between CP+30B and CP+57B, so that they appear in the B-display. As the action of these subroutines is logically separate from the rest of 1IQ, any further description is left to the section under "Subroutines".

Now 1IQ looks for requests from 1FM or 1PL to terminate a file. If one is found, it is assumed that 1FM or 1PL has already finished its actual output, and may have dropped its PP, but it should have set bits 48-59 of the first word of the FET to 7777B as a signal to 1IQ to drop the equipment from the control point, release the buffer, zero the FNT entry and the FET, and release the disk space occupied by the file.

If bits 18-23 of the input register are r and bits 12-17 are s, then r+1 is the number of the first FET for film or plot, and s is the number of the last one. If r=s, there are none, and control passes to KA. Otherwise bits 48-59 of the first word of each FET numbered r+1 through s inclusive are tested; if they = 7777B, the following steps are carried out: (1) Find the buffer number. The FIRST pointer in the FET gives is starting address; subtract from this the address of buffer 0, and divide the result by 400B, 1000B or 2000B (the length of a buffer). To release this buffer, call subroutine FETCH; zero BUFAS+n (the buffer number); reduce KEY+3 (the count of buffers now assigned) by 1; and call subroutine PUT. (2) The FNT address is in bits 0-11 of the sixth word of the FET (this is true only of FET's for 1PL and 1FM); read the FNT entry, set up a stack request to release its disk space, and call resident subroutine R.EREQS to issue the request. (3) Zero the FNT entry. (4) Zero the FET. (5) In the copy of the FET still present beginning at MESS, get the unit EST ordinal from bits 48-59 of the 15th CM word; release the equipment from the control point.

When any termination requests from 1PL or 1FM have been dealt with, 1IQ comes to KA to deal with the possibility of expanding and contracting the field length. First it calls subroutine FETCH, as buffers may be assigned or buffer counts altered.

Now at KAFA, an attempt is made to satisfy requests for additional buffers. Control point +73B contains 1 in bits 48-59 if 1IR has requested a new buffer (it can assign itself buffers, thanks to the interlock, but cannot expand storage if there is no free buffer). The same byte will be set to 2 by 1IQ if it pre-assigns a buffer to 1IR, in which case bites 0-11 of the same word will be set to the buffer number. If 1IR accepts the buffer, it sets bits 48-59 of CP+73B back to 0. The word at CP+74B is used in the same way by 1IQ to request a new buffer

from itself for use by 1PL or 1FM. This roundabout self-request is used because the need is seen in a different part of 1IQ, and it is simpler to communicate it to the satisfaction part of 1IQ in such a way that the 1IR-satisfying routine can handle it as well. First these two cells in the control point are checked, and a count is made (in D.BA+4) of how many requests they contain. The answer could be 0, 1, or 2. If 0, go to KAC; otherwise try to satisfy the 1 or 2 requests using existing field length, and then go to KAC leaving the number of still-unsatisfied requests in D.BA+4. (KEY+1)-(KEY+3) is the number of unused available buffers. If this is 0, the only possibility is to go to KAC; otherwise, go once or twice through the loop between KAB and KAF. If a request can be satisfied by buffer number n, replace the request word at CP+73B or 74B by VFD 12/2,36/0,12/n; set BUFAS+n to 1000B for a request from 1IR or 2000B for a request from 1IQ; and add 1 to KEY+3, the count of currently-assigned buffers. (The values 1000B and 2000B have no meaning except for being non-zero. A byte in the BUFAS area is normally set to the FET number when a buffer is assigned to the FET, but again, this number has no significance except for being non-zero.)

If there are no requests, or when as many requests as possible have been filled from existing field length, we come to KAC with the number of unfulfilled requests in D.BA+4. If this is 0, expanding storage has been taken care of and we go to KC to work on the possibility of shrinking storage. Also, if 1IQ found the error flag set when it was called, instruction BRWD has been set to a no-op, and we go to KC without trying to expand storage. Otherwise, we convert the number of wanted buffers to their length, and see from CMR cell T.UAS whether so much free storage exists. If not, pass on to KC. If so, request the storage, pause, and see if granted. If not granted, return to KAK to recheck T.UAS and repeat the cycle. If granted, adjust D.FL and also the field length in bits 0-11 of the input register (because those bits are compared with the field length every time 1IQ is recalled) and then call FETCH to enable buffer assignments and counts to be altered. To KEY+1, the count of buffers in the whole field length, add the increase in the field length divided by 4, 10B, or 20B, a buffer length/100B. Then read the clock and reset KEY to the current time in seconds +15; this is done whenever field length is increased, and whenever the possibility of shrinking field length is considered or implemented, and it serves as an interval timer for considering that possibility. Then return to KAFA to pick up the requests again and try to satisfy them with the new field length.

KC is reached when everything necessary or possible has been done about satisfying requests for additional buffers. Now it may be time to think about shrinking storage; if not, branch straight to BRWD. The word at control point +77B is read (the interlock has not been secured through subroutine FETCH, but no matter, as 1IQ is not altering anything in CP+77B through 104B, just reading the timer in one of them.) If the interval timer is above 3777B, while the seconds clock is below 100B, presume that the clock has passed the timer and go to KCC. (Note that both are 12-bit counters.) Or if the clock is above the timer, go to KCC, otherwise to BRWD. At KCC, call FETCH as the number of buffers may be changed. Compare the number of buffers in the field length with the highest number in use at any one time in the last 15 seconds (since the last time this was done.) If two or more buffers have gone unused it may be possible to shrink field length; otherwise go to KDZ. Now in BUFAS through BUFAS+n-1 there is a register of assigned and free buffers; scan it backwards from BUFAS+n-1 towards BUFAS until an assigned buffer is found, or until bytes indicating all but one of the free buffers have been scanned. This gives the number of buffers that can be dropped. If 0, go to KCE - it would appear that the program could just as well go straight to KDZ here. If non-zero, multiply the number by BUFLG/100B to get the amount to reduce field length, set up a request, and keep requesting until the field length is changed, when it is safe to assume the request was exactly fulfilled. Then at KCE pause, adjust D.FL and bit 0-11 of the input register, which must always be equal to it, subtract the number of buffers dropped from KEY+1, the counter of buffers in the whole field length, and pass to KDZ.

We reach KDZ whenever the time has arrived to try to shrink storage, and any possible shrinkage has been accomplished. We set KEY+2, the counter of the largest number of buffers recently in use, equal to KEY+3, the counter of buffers currently in use, as we are beginning a new 15-second interval. Then read the seconds clock and set the interval timer in KEY to 15 seconds after the current time (mod 4096). Then call PUT, and pass on to BRWD.

We come to BRWD when finished with expanding or shrinking storage. It is possible that 1IQ will now find itself a buffer short, in attempting to set up a file for 1FM or 1PL; but in that case it will not re-open the field length question, but store a request to itself in control point+74B, to be worked on two seconds later the next time 1IQ recalls itself. If 1IQ found an error flag when it was called, the switch at BRWD has been set to skip the possibility of finding new film or plot files, and branch straight to FNT7. Otherwise, between BRWD and BRWDC, scan the EST for any free film, plot, or hard copy equipment (codes FM, PL, and HC) that are not off. If there is none, go straight to FNT6, as there is no possibility of dealing with any new files for such units. If any such unit is found, go to BRWDC, and scan the FNT for entries at control point 0, not locked out, on allocatable devices, not assigned to EXPORT-IMPORT or RESPOND,with disposition codes between 20B and 37B. Having exhausted the FNT, we come to FNT5 with, in FM, the address of the highest-priority FNT entry whose disposition code is between 20B and 23B (film), or 0 in FM if there is none such; in HC, the address of the highest-priority FNT entry whose disposition

code is between 24B and 27B (hard copy), or 0 in HC if there is none such; and in PL the address of the highest-priority FNT entry whose disposition code is between 30B and 37B (plot), or 0 in PL if there is none such. Now call subroutine FNTH with each of these addresses. If there are files for 1IQ, it will claim them by changing their control point numbers, and the FNT pseudo-channel can now be dropped, which was requested just before starting the scan of the FNT. Next call subroutine FNT8 three times, to bring together if possible (1) a film unit, film disposition file, FET, buffer, and PP with program 1FM; (2) a plot unit, plot disposition file, FET, buffer, and PP with program 1PL; and (3) a hard-copy unit, hard-copy disposition file, FET, buffer, and PP with program 1FM. Then pass to FNT6.

Control comes to FNT6 when film and plot possibilities have been dealt with, or it has found that there are no such units in the EST anyway. Bits 48-59 of control point +76B are initially 0; 1IQ sets them non-zero just before calling 1IR to another PP, and 1IR sets them back to zero just before dropping itself. So now this byte is checked; if non-zero, 1IR is already active and control goes straight to FNT7 to drop 1IQ after a final passage through subroutines CONSOLE and STATUS. Otherwise, 1IR is not active; it will be called in a minute (1IR must come up every so often, as no other program can check the card reader(s) to see if the operator is trying to feed it/them). But first, if a buffer has been preassigned to 1IR on its request, and 1IR has not accepted it, and 1IR is not active, the buffer should be de-pre-assigned. (1IR could request an additional buffer and then satisfy its need when a file terminated, without ever picking up the additional buffer. A bit later, 1IR might drop, and if all equipments remained inactive for a long time 1IR would still never pick up the additional buffer. If it happened to be the highest-addressed buffer in the field length, this might prevent 1IQ from shrinking field length, even though in fact nothing at all was being done by JANUS.) So FETCH is called, and CP+73B read. If bits 48-59 do not = 2, there is no pre-assignment to 1IR and control can pass on to FNT6A. If there is a preassignment, the number of the buffer (n) is in bits 0-11, so zero BUFAS+n to cancel the assignment, zero CP+73B, and reduce by 1 the count of buffers assigned in KEY+3. Then, at FNT6A, call PUT; set bits 48-59 of CP+76B non-zero, to show that 1IR is active, and call 1IR.

Finally, at FNT7, subroutines CONSOLE and STATUS are called again to deal with operator type-ins and equipment messages, and then at PPREC 1IQ is dropped with a 2-second recall.

Subroutines

FNTH

This brings a file to our control point from control point 0. It is entered with the address of the FNT entry in the A-register; however, if this address is 0, FNTH does nothing.

Entry Information

The FNT address, or 0, in the A-register.

Exit Information

None.

Subroutines Called

None.

Registers Destroyed

D.Z1, D.T0 through D.T4

FNT8

Calling Sequence: RJM FNT8
VFD 12/a,12/b,12/c

This subroutine may assign a film or plot disposition file to a PP and an equipment, an FET and a buffer. "a" in calling sequence is the address of a direct cell (FF+2 or FF+3) that contains 0 if no file, or the address of the FNT entry of the file. "b" is display code FM or PL, for requesting an equipment of one type or the other. "c" is the address of a call to 1FM or 1PL, which is followed by the word "FILM" or "PLOT" to be used in giving a new name to the file.

First the exit address is set in FNT8X; then the cell the first parameter points to is checked. If the cell contains zero, exit immediately. Otherwise, get r and s from bits 18-23 and 12-17 of the input register; r+1 is the number of the first film/plot FET, and s is the number of the last. If r=s, there are no such FET's, and control goes to FNT8Y to release the file from our control point and exit FNT8. Otherwise scan the series of FET's for a free one, as defined by bits 48-59 of its first word being 0. If none is free, go to FNT8Y as above.

If an FET is free, come to FNT8B with its number in D.BA+2. Take "FM" or "PL" out of the calling sequence, set up a request for an equipment of that type, and issue the request. If the equipment is not obtained, go to FNT8Y as above. Otherwise, save its EST ordinal in D.BA+3. Call FETCH, as a buffer must now be assigned to the file. Also read the word at control point +74B. If bits 48-59 of this word = 2, the routine has pre-assigned a buffer to itself on a previous recall of 1IQ, and its number is in bits 0-11 of the same word. So zero that

word in central memory, and then go to FNT8E with the buffer number in D.Z7. If such a preassignment is not indicated in CP+74B, begin looking for a free buffer at FNT8H. If the count of buffers in the field length is not greater than the count of buffers in use now, there is no free buffer, so set up a request for a new one (set bits 48-59 of CP+74B=1; this will be picked up by 1IQ the next time it recalls itself, at KAFA in the main program); call PUT to release the buffer assignment registers; release the film or plot equipment, and branch to FNT8Y as above. But if the counts show there is a free buffer, go to FNT8K to search BUFAS and following to find it. On finding it, add 1 to KEY+3, the count of buffers now in use, and pass to FNT8E with the buffer number in D.Z7.

At FNT8E, claim the buffer by putting the FET number in its BUFAS slot (if the buffer was pre-assigned, this number will replace not zero, but 2000B; however, only zero and non-zero actually signify). Now an FET is set up in the PP memory beginning at MYFET. From the buffer number in D.Z7, and the starting address of buffer zero, the starting address of the new buffer is calculated, and hence set up the four FET pointers. From the third parameter in the call to FNT8, the address of the call to 1PL or 1FM is found, and into this call is inserted the EST ordinal of the equipment, in bits 24-35, the disposition code of the file, in bits 12-23, the starting address of the FET, 20B times the FET number, in bits 0-11 of the call. Next a new name is given to the file, both in the FNT entry and in the FET. This is necessary because the present name of the file is the same as the name of the job that generated it; if the job generated one film and one plot file, they now have the same name, and if 1FM and 1PL each call CIO to read one of the files, its name in the FET will not be a sufficient identification. So take the word "FILM" or "PLOT" from the area immediately after the call to 1FM or 1PL, add after it a letter corresponding to the number of the assigned FET (A=1, B=2 etc.) and put the resulting unique name into the first word of the FET and the FNT entry. (The name is unique as far as this control point is concerned, which is enough.) The previous file name (job name) is saved in bits 18-59 of the sixth word of the FET; the address of its FNT entry is stored in bits 0-11 of the same word. The status in the FET and FNT is set to 3, rather than to 53B, for rewound, which would seem the logical value. This is because the file may at some point be rewound and reprocessed by 1PL or 1FM, and 1PL or 1FM should be able to distinguish easily between the status immediately after such a rewind, and the initial status, marked here by 3. The EST ordinal of the device is stored in bits 48-59 of the 15th word of the FET. The rest of that word is initially zero; 1IQ uses bits 48-59 to recognize that this is the FET to which an operator type-in naming that device is to be directed, and 1IQ puts a number corresponding to the type-in into bits 36-47, which should be zeroed by 1FM or 1PL when it accepts the message.

The display code letters "HC", "FM", or "PL", followed by the EST ordinal as a 2-digit octal number in display code, are put into bits 36-59 of the 8th word of the FET. Two blanks are put into bits

24-35 of the same word. The former name of the file has already been saved, as well as in the sixth word of the FNT, in bits 0-23 of the 9th word and bits 24-59 of the tenth word; bits 24-41 of the latter are now zeroed to get rid of whatever was originally in bits 0-18 of the first word of the FNT entry. The rest of the 10th through 14th words of the FET are initially zero. This makes the six words between the 9th and 14th words into, for example: FM26 JOB001A with 23½ bytes of zeros after the job name. As soon as the FET model is written to its destined space in central memory, these six words will become a line in the J-display by DSD. The first full byte after the job name (bits 24-35 of the 10th word of the FET) will be periodically checked by 1IQ, in subroutine STATUS, and if they are non-zero it will be assumed that the whole six words now form a display message, which will be copied to the control point area so as to form a line in the control point B-display. Thus 1FM or 1PL can send a message to the operator by putting it into bits 0-35 of the 10th word and all of the 11th through 14th words, being sure to make bits 24-35 of the 10th word non-zero. To drop the message from the control point B-display, 1FM or 1PL need only zero bits 24-35 of the 10th word, but obviously the rest of this area of the FET should be zeroed as well for simplicity.

Finally the 16-word FET is copied into its area in central memory, and subroutine CALL is called to issue the call for 1PL or 1FM. Then subroutine FNT8 exits. The FET looks like this:

```
VFD 24/0HFILM,6/n,30/3   (or 24/0HPLOT etc.)

VFD 12/0,12/0400B,36/a,60/a,60/a,60/a+BUFLG

VFD 42/jobname,6/0,12/b

VFD 60/0,60/0

VFD 12/0HFM,12/dd,12/5555B,42/jobname,42/0  (or 12/0HPL etc.)

VFD 60/0,60/0,60/0,60/0

VFD 12/e,48/0,60/0
```

where n is the FET number, a is the address of the start of the buffer, "jobname" is the name of the job that generated the file (necessarily 7 characters), b is the address of the FNT entry for the file, dd is the EST ordinal of the device, as two octal digits in display code, and e is the same number as a binary integer. The call to 1PL or 1FM is:

```
VFD 18/0H1PL,3/0,3/c,12/e,12/g,12/f   (or 18/0H1FM etc.)
```

where c is the control point number, e is the EST ordinal of the device as a binary integer, f is the address of the start of the FET, and g is the disposition code of the file.

Subroutines Called

CALL, FETCH, PUT, R.PROCES

Registers Destroyed

D.BA through D.BA+4, D.TO through D.T4, D.Z1, D.Z2, D.Z3, D.Z7 directly; D.Z4 and D.Z5 by FETCH.

CDEV

This scans the EST and counts the number of card readers, up to a maximum of 4; punches, up to a maximum of 3; printers, up to a maximum of 12; and film, plot and hard-copy units, up to a maximum of 4 in all, and leaves the respective totals in D.Z1, D.Z2, D.Z3, and D.Z4. Units are counted whether assigned at present or not and whether on or off.

Entry Information

None.

Exit Information

The counts in D.Z1 through D.Z4.

Subroutines Called

None.

Registers Destroyed

D.Z5, D.TO through D.T4, D.BA through D.BA+4.

CONSOLE

This transfers and translates operator type-ins from the control point area where they are put by DSD to the proper FET's, where they can be obeyed by 1IR, 1FM and 1PL.

The nine words in control point +60B through control point +70B are used as a first-in, first-out stack for type-ins. DSD inserts a type-in at CP+60B if this word is zero; if not, it refuses the type-in and comments: "TRYLATER". A type-in has the form /END07. where "END" is the particular command, and 07 is the octal EST ordinal of the device, which identifies

the file now being processed on that device. DSD will have cut off the "/" and put the "END06" into CP+60B, left-justified with zero fill. (The slash is used to identify type-ins for JANUS, so that DSD need not contain a list of the allowed commands, though at present it does contain one.)

CONSOLE first reads the nine-word area into the 45 bytes beginning at MESS. Now if the ninth word of the area does not begin with a zero byte, i.e. is not empty, the stack is full and a new type-in could not be accepted, so we go straight to CONSOLP. If the first word, corresponding to CP+60B, is zero, there is no new type-in to be accepted, and we go straight to CONSOLP. Otherwise, CP+60B, the new type-in, is copied to the first empty slot at the end of the stack in MESS+5 ff., and then we zero CP+60B to show that the type-in has been accepted. This brings us to CONSOLP.

At CONSOLP, where we arrive whether or not a new type-in has just been accepted and put at the end of the stack, a check is made of the word at the head of the stack in MESS+5 through MESS+9. If MESS+5 contains zero, the stack is empty, and we exit from CONSOLE without copying the stack back to the control point. (This is not necessary because the stack must have been empty before we read it from the control point and a new type-in must not have been added to it.) Otherwise try to match the word with the list of messages beginning at OMES. To be acceptable, the type-in in MESS+5 through MESS+9 must match one of the messages character-by-character from the left until an empty character in the model is reached. At CP, the 10 characters of the type-in are spread out into BUFAS through BUFAS+9, to make the comparison easier. At CONSOLB, PL is set to point to the beginning of the next word in table OMES; if it points to a zero byte, the table has been exhausted without finding a match and we branch to CONSOLQ, as if we had found a match and passed on the message. Otherwise, PL is copied to HC, FM is set to point to BUFAS, the beginning of the spread-out type-in, and the comparison is begun.

Each comparison then begins at CONSOLC. When the first zero character in the model is found, we go to CPA. Otherwise, if a mismatch is found, we go to CONSOLB to try the next entry in table OMES. At CPA, we see if the type-in is "/ABORT." If so, put out a dayfile message; then drop the PP in such a way as to set the error flag to F.ERPP. 1IQ will not be recalled, and as soon as 1IR sees that flag, it will drop itself and so drop JANUS. Otherwise, go to CPR and pick up the next character, after the word. This must be the first of one or two octal digits giving the EST ordinal of the device, identifying the file involved. If not an octal digit, go back to CONSOLB to try the next entry in table OMES. (This allows for the unlikely possibility that the table contains two words of which the later one is an overhang of the earlier one, such as BS and BSP.) If this character is an octal digit, the next character of the type-in must be another octal digit, or a period, comma, or blank; otherwise there has certainly been a bad type-in and control

goes to CONSOLQ. Now set D.Z5 to contain the binary form of the octal number. Now initialize D.Z7 at 0; if there are any more parameters in the type-in, they will be stored in it. At present, this is only possible for the BS type-in (backspace).

/BS07 is equivalent to /BS07,1,1.
/BS07,10. is equivalent to /BS07,10,1.
/BS07,,A. is equivalent to /BS07,1,A.

In general, the first parameter after BS07 is a decimal integer (interpreted modulo 64, with 0 considered to mean 1) giving the number of pages to be backspaced. The default value is 1 page. The second parameter is the character, which must be a letter or digit, that is to be regarded as the format character defining the beginning of a new page; the default value is display code 1, which is the character normally used for this purpose.

Now the next character after the equipment EST ordinal is picked up. If this is not a comma, there are no more parameters and control goes to CPB. Otherwise, beginning at CPD, pick up the characters after the comma and accumulate them as decimal digits into a binary total in bits 6-11 of D.Z7. As soon as a comma is reached, control goes to CPC to get the page format character. Or if any character that is neither a decimal digit nor a comma is met, it goes straight to CPB assuming there is no format character. At CPC the character that follows the comma after the page count is picked up. If it is not a letter or digit, go to CPB; otherwise store it in bits 0-5 of D.Z7 and arrive at CPB. At CPB, PL still points to the start of the entry in table OMES that was matched. The fifth byte of that entry, the number corresponding to the word of the type-in, is now stored in D.Z6. Now get the total number of FET's (active or not) from bits 12-17 of the input register, and look at the 15th word of each FET to see if its bits 48-59 match D.Z5. If no match, go to CONSOLQ. If there is a match, go to CONSOLH to see if bits 36-47 of the same 15th word are zero. If so, go to CONSOLW to insert the message there. If not, go to CONSOLY. 1IR, 1PL, or 1FM, in charge of that FET, has not yet accepted the last message that was left there. However, if the new message is /REW, it is allowed to override any waiting one and we go to CONSOLW after all. A stack of up to 8 type-ins is waiting in MESS+5 through MESS+44, and the first one is a hang-up. So zero MESS+45 through MESS+49, representing a 9th slot; then find the first empty slot beginning at MESS+10; then copy MESS+5 through MESS+9 into the empty slot; then write MESS+10 through MESS+49 back to control point +61B through control point +70B and exit from CONSOLE. No attempt is made to process the next type-in in the stack, as this might bring us back to the hang-up, and 1IQ must be kept in motion.

At CONSOLW, the message number is inserted in bits 36-47 of the 15th word of the proper FET, and control comes to CONSOLQ.

At CONSOLQ a translated type-in has been passed successfully to its FET, or a type-in has been abandoned because the message is not in table OMES or because the unit it named is not names by any of our FET's. The first word in the stack (MESS+5 through MESS+9, corresponding to CP+61B) has been disposed of; the stack is now shifted down by zeroing MESS+45 through MESS+49 and copying MESS+10 through MESS+49 into CP+61B through CP+70B. Then control returns to the beginning of subroutine CONSOLE to accept a new type-in, if any, and process the next item in the stack, if any. The fact that this is repeated will not delay 1IQ unduly, for obviously, as long as a message is not blocked by an earlier one in the same FET (in which case control goes to CONSOLY), the stack is disposed of much faster than the operator can add to it by typing in.

Strictly speaking, the nine-word stack begins at CP+61B, runs to CP+70B, and ends with CP+60B, corresponding to MESS+5 through MESS+44, and ending at MESS through MESS+4. When read is performed from the CP area into the MESS area, the last item (CP+60B) is read again, unless the stack is full, into the first free slot in the range MESS+5 to 9 through MESS+40 to 44. The message at the head of this having been dealt with, MESS+45 through MESS+49 are zeroed, and the first to eighth items in the stack are considered to be MESS+10 through 49, which are replaced in CP+61B through CP+70B. At CONSOLY, the stack is rotated one place forward, end-around, rather than the eighth item being shifted forward and zeroed, before replacing MESS+10 through 49 in CP+61B through CP+70B.

Entry And Exit Information

None relevant to other parts of 1IQ.

Subroutines Called

None.

Registers Destroyed

D.Z1 through D.T4

STATUS

This copies display messages from the 9th through 14th words of up to 4 FET's into the four areas at control point +30B through 35B, CP+36B through 43B, CP+44B through 51B, and CP+52B through 57B. These areas are treated by DSD as the four variable lines of control point B-display, because bit 12 of the control point +153B is 1.

In an inactive FET, all the words are zero, and STATUS will do nothing about them. In an active FET, the 9th through 14th words are treated as 30 bytes, in which the first contains "CR", "CP", "LP", "FM", or "PL", depending on the type of equipment working on the file; the second contains the EST ordinal of the equipment, as a two-digit octal number in display code; the third contains two blanks; and the fourth through seventh contain the name of the job that generated the file. If the eighth byte is 0, it is assumed that the rest of the 23-byte area following the job name is also empty. Then DSD will display the first seven bytes as a line in the J-display, but nothing will be copied to the B-display. If the eighth byte is not zero, it is assumed that there is a message beginning with the eighth byte and running to the first zero byte, or to the 30th byte if there is no zero byte. Then DSD makes everything up to the end of the message into the line in the J-display, and subroutine STATUS makes it into a line in the B - display (unless four lines have already been put there, filling it).

First set D.Z1 to 30B, and D.Z2 to 60B, indicating that the available space for the B-display runs from CP+30B to CP+57B. Set D.Z4 to 30B, indicating that the message area of the first FET begins at RA+30B. From bits 12-17 of the input register, get the number of FET's, and store it in D.Z3. Then work through the FET areas, copying the 8th to 14th words of each into MESS through MESS+29. If MESS+7 is 0, pass over an FET. Otherwise go to STATUSB, where blanks are put in MESS+5 and MESS+6 if they contain zero, and then copy the six CM words into the next available slot in the control point area. (Those two bytes will not contain zero if the file at the FET was generated by a normal job; but when an output file is generated from the console by DIS, for instance, its name will simpy be "DIS", and in that case the bytes will contain zero.)

An exit is made from STATUS when all four spaces in the control point area have been used, or there are no more FET's. In the latter case, the unused part of the control point B-display area is zeroed.

Entry and Exit Information

None relevant to other parts of 1IQ.

Subroutines Called

None.

Registers Destroyed

D.Z1 through D.Z5, D.T0 through D.T4

CALL

This is used to call 1IR, 1IS, 1FM and 1PL as necessary, using the M.RPJ function with zero delay.

Entry Information

The address of the PP call to be put in the message buffer is in the A-register.

Exit Information

None.

Subroutines Called

R.PROCES

Registers Destroyed

D.TO through D.T4.

FETCH

This is used to get access to CP+77B through CP+104B, which contains a timer, counts of buffers, and a register of buffer assignments. As 1IQ and 1IR both have to be able to change them, and may both try at once, the following interlock is used:

Let there be n competitors for access, numbered 1 through n. There are n CM cells designated B(1) through B(n), n more designated C(1) through C(n), and one CM cell designated K. Initially, all are zero except K, which must contain the number of any competitor, not 0. (1IQ initializes it to 1, its own number, below BRWBA.) To secure access, a competitor performs these steps:

1. Set B(i) non-zero.
2. If K contains i, go to step (5). Otherwise it contains k.
3. Set C(i) zero. If B(k) is non-zero, return to step (2).
4. Set K to contain i, and return to step (2).
5. Set C(i) non-zero. If any other C is now non-zero, return to step (2). Otherwise, competitor i has access, and others are locked out. To relinquish access, a competitor sets B(i) and C(i) to zero.

FETCH and PUT in 1IQ and 1IR use this scheme. Note that 1PL and 1FM do not need to use it, as 1IQ assigns buffers to them before

calling them, and releases their buffers on their request, made by setting bits 48-59 of the first word of the relevant FET to 7777B. As there are only two competitors, "i" does not need to be treated quite generally. For convenience (simply because there is a constant 3 in a direct cell in program 1IR, but not a constant 2) in 1IQ considers its "i" to be 1, while 1IR's is 3. Then B(1) is CP+105B, B(3) is CP+106B, C(1) is CP+107B, C(3) is CP+110B, and K is CP+111B. Only bits 48-59 of each cell are significant.

Note that at the beginning of step (2) above, FETCH always calls subroutine PAUSE, as 1IQ might be locked out by 1IR at a time when 1IR is requesting a monitor function.

When FETCH has secured access, it copies CP+77B into the five cells beginning at KEY , and CP+100B through CP+104B into the 25 cells beginning at BUFAS, and exits with the address CP+105B in the A - register; which content is often used to save one or two instructions in reading another control point word immediately after calling FETCH.

Entry Information

None.

Exit Information

The address CP+105B in the A-register, and KEY and BUFAS areas set up.

Subroutines Called

PAUSE.

Registers Destroyed

D.Z1 through D.Z5 directly, D.T0 through D.T4 by PAUSE.

PUT

This releases the interlock made by the FETCH subroutine, and returns the KEY and BUFAS areas to CP+77B through CP+104B, after setting KEY+2 equal to KEY+3 unless KEY+2 already contains a number equal to or greater than that in KEY+3. This maintains bits 24-35 of CP+77B as a register of the largest value bits 12-23 have reached since the last time the possibility of shrinking field length was looked at.

To release the interlock described above in subroutine FETCH, competitor number i, currently having access, sets B(i) and C(i) to zero; i.e. 1IQ zeros bits 48-59 of CP+105B and bits 48-59 of CP+107B.

Entry and Exit Information

None.

Subroutines Called / Registers Destroyed

None.

PAUSE

This is merely a more convenient way of calling the PP resident subroutine R.PAUSE. First we call it, then reset D.RA, whose new value is in the A-register on return, and then check the control point error flag. We go to DROP if it is F.ERPP. If it is zero or anything else, we exit from PAUSE. The situation is parallel to that described above at the beginning of the 1IQ narrative.

Entry and Exit Information

None.

Subroutines Called

R.PAUSE

Registers Destroyed

D.T0 through D.T4

CALL1IS

This calls program 1IS to another PP, either to get the total length of the overlay library stored in bits 0-11 of RA+17B, or to get the overlays stored in the field length and the directory to them stored in RA+10B through RA+17B. First we call subroutine CALL to call 1IS (the choice between the above two possibilities depending on whether R1IS+4 contains 0 or the address of the first word after the last FET.) Then wait one millisecond and see if bits 48-59 of CP+76B contain 0. If so, 1IS has not finished its task and CALL1IS goes back to wait another millisecond. If not, 1IS is finished; we zero bits 48-59 of CP+76B and exit from CALL1IS.

Entry Information

The call to 1IS set up on one of the two possible ways, at R1IS.

Exit Information

Either the length of the overlays in RA+17B, or the overlays and their directory properly set up in the field length.

Subroutines Called

CALL, PAUSE; R.PROCES by CALL; R.PAUSE by PAUSE.

Registers Destroyed

D.T0 through D.T4.

16.1.2 1IR General

1IR is a PP program, always loaded at 1000B. It is called only by 1IQ; 1IQ recalls itself every 2 seconds, and if it finds that 1IR is not already at the control point, it calls 1IR.

As there is not room in the PP memory for the logical entirety of 1IR, which drives up to 19 card readers, punches, and printers at the JANUS control point, several parts have been broken off and treated as overlays, which are loaded from central memory into the area between 1000B and 2000B, and executed there. For simplicity, these overlays are originally stored in central memory by the PP program called 1IS; but as they have no meaning outside the context of 1IR, and cannot function unless along with it, the overlays are described here under the heading 1IR although they exist in the library as part of 1IS.

Each overlay begins with a CM word

```
VFD 12/a,12/b,12/c,24/0
```

where a is the length of the overlay in CM words, not including this initial word; b is the address in the 1IR PP, beginning at which the overlay (not including this initial CM word) is to be loaded, and c is 0 if the overlay is to be entered as a subroutine, and 1 if not. Note that overlays that can be called as subroutines by other overlays are loaded as high as possible in the range 1000B to 2000B, while other overlays are loaded at 1000B.

1IR itself fills the PP memory with program from 1000B to just before a cell multiply named as IMAGE, BUF, BUFAS, and MYFNT (except that 1000B to 1777B, which have to be left free for overlays, initially contain nothing but a jump from 1000B to the preset routine, and BSSZ in the rest.). 160 cells beginning at IMAGE are used as working storage for various purposes, and are also occupied initially by the preset routine. So if the program is modified, it must not expand beyond the point at which IMAGE+160=7777B. If it does, a warning error will be produced by a conditonally-assembled instruction just before PRESET.

Function

1IR does all the work of the JANUS control point (combination of READ control point and OUTPUT control point) except:

1. changing field length (done by 1IQ)
2. picking up type-ins from where DSD has left them, and putting B-display messages where DSD will pick them up (done by 1IQ)
3. backspacing print files (done by 1IU)
4. assigning output files to film, plot, or hard-copy units (done by 1IQ)
5. driving film, plot, or hard-copy units (done by 1PL, 1FM, etc.)

Entry Information

The input register contains

VFD 18/0H1IR,3/0,3/c,6/p,6/q,6/r,6/s,12/w

where c is the control point number, p is the number of the highest numbered FET for a card reader (they are numbered 1 to p), q is the number of the highest-numbered FET for a card punch (they are numbered p+1 to q), r is the number of the highest-numbers FET for a printer (they are numbered q+1 to r), s is the number of the highest numbered FET for a film, plot, or hard-copy unit (they are numbered r+1 to s, or r=s if there are none), and w is the address of the beginnirg of buffer number 0.

Control point +77B contains

VFD 12/a,12/b,12/c,12/d,12/0

where a does not concern 1IR; b is the total number of buffers in the current field length; d is the number of buffers now assigned; and c is a counter that does not affect 1IR, but that 1IR must set equal to d whenever it increases d and finds d then larger than c. (1IQ, every 15 seconds, takes c to be the highest number of buffers that have been simultaneously assigned since the period began; having perhaps reduced the number of buffers, but not below c+1, 1IQ sets c=d and begins a new 15-second period). CP+100B through CP+104B contains 25 bytes of which the first 23 are used as buffer assignment registers, containing zero if a buffer is free, and non-zero if assigned. CP+77B through CP+104B must not be altered by 1IR unless it has secured permission and set the interlock against 1IQ by calling subroutine FETCH. On FETCH and PUT (which releases the interlock), see the practically identical subroutines in the documentation to 1IQ.

Exit Information

Just before dropping its PP, 1IR zeros CP+76B to inform 1IQ that 1IR is no longer active.

Other Programs Called

2TJ is called as an overlay and loaded at 1000B. 1IU is called to another PP to backspace a print file when this is requested by a type-in. The overlays that 1IS has stored in JANUS field length are variously called and loaded somewhere between 1000B and 2000B; but they are loaded by subroutine EXECUTE rather than by any sort of program calls.

FET Formats

Each FET occupies 16 CM words, of which the first 5 have the form of an ordinary minimum-length FET with the EP bit set =1 so that a disk parity error will not abort JANUS. Whenever 1IR is working on a file, the first word of its FET is first read into CURFET through CURFET+4 (for convenience in looking up cross-references, CURFET+3 and CURFET+4 are addressed as CURFET3 and CURFET4.) The FIRST pointer address is copied into subroutine FIRST; the difference between the IN and FIRST pointers is stored in byte IN, and the difference between the OUT and FIRST pointers is stored in byte OUT. The difference between the LIMIT and FIRST pointers is a constant, and can be found at LIMIT,TYPE when TYPE contains 0 for punch files, 1 for printer files, and 2 for card reader files.

The 6th, 7th, and 8th words of an FET are similarly read into D.FNT through D.EST+4. The cells that contain the 8th word are referred to, for cross-reference convenience, as DEST0, DEST1, DEST2, DEST3, and DEST4. The 6th word, for a print or punch file, is entirely zero (reserved for future system use.) For a card read file, the first and second bytes of this word are referred to simply as D.FNT and D.FNT+1; the 3rd, 4th, and 5th bytes are referred to as RCT, CCT, and RTYPE respectively, as they contain the record count within the file, the card count within the record, and the current record type (binary, BCD, or special.)

The 7th word of the FET, which is read into D.FNT+5 through D.FNT+9, is the most complicated. For all kinds of file, D.FNT+8 and D.FNT+9 are referred to as CT and CT+1, and contain a 24-bit count of the number of cards read or punched, or of lines printed, in the file so far. For a punch file, D.FNT+7 is referred to as PCT, as it contains a count of the number of cards punched so far in the current record. For a read file, this byte is referred to as G7, and contains the right 12 bits of the 15-bit time limit on the job card. For a print file, this byte is not used (reserved for future system use.)

The left two bytes of the 7th word of the FET are read into D.FNT+5 and D.FNT+6, which are referred to as F5 and F6 for a print or punch file, and as G5 and G6 for a read file. These bytes contain a number of one-bit flags, referred to as FL.IMAGE, FL.OS, and so on. For instance, if the leftmost bit of F5 is a flag called FL.OS, FL.OS is defined as the quantity 4000B, which can mask out this bit. A macro called TESTBIT is defined so that TESTBIT FL.OS; i.e. TESTBIT 4000B; will be assembled as SHN 6; i.e. the shift needed to bring the flag bit, just after the byte is loaded into the A register, to the sign bit position. So a test for flag OS could be made with either LDD F5; LPC FL.OS; NJN YES; or LDD F5; TESTBIT FL.OS; MJN YES. After TESTBIT, the macro RETEST can be used to test another flag bit in the same byte. Suppose the bit next right of flag OS is flag PROS; FL.PROS is defined as 2000B. Then RETEST FL.OS,FL.PROS will assemble as SHN 1, i.e. the shift needed to rotate the A register from the position in which flag OS is in the sign position to that in which flag PROS is in the sign position. RETEST FL.PROS,FL,OS would assemble as the opposite shift, SHN 17. So the sequence:

```
LDD       F5
TESTBIT   FL.OS
MJN       AAA
RETEST    FL.OS,FL.PROS
MJN       BBB
```

will branch to AAA if flag OS is 1, or to BBB if OS is 0 but flag PROS is 1.

Here is a list of these flags (bit numbers are for the CM word as a whole).

In G5:
Bit 59, FL.WTFNT, is 1 when a read file is ready to be terminated but a pre-OUTPUT file has to be written containing error messages generated by card reading, and an FNT slot has to be come free before this file can be set up and written, after which we shall complete the termination of the job file.

In G6:
Bit 47, FL.EOJ, is 1 when a read file has ended with a 6-7-8-9 card not preceded by a 7-8-9 card; the reading of a 7-8-9 card must be simulated, but the next time we come around to this file FL.EOJ will tell us to simulate the reading of a 6-7-8-9 card rather than physically reading another card.
Bit 46, FL.PREAB, is set then a fatal error is found in an input file, i.e. a binary card with a bad checksum or with an improper format. Once this is set, we continue reading and checking the rest of the cards in the file, but do not bother writing them to disk; and at the end of the file we set a flag in its FNT entry that will tell 1RA to abort the job as soon as it comes to a control point.
Bit 45, FL.JOBER, is set to 1 if 2TJ indicates that the job card contains an error. This has the same effect as FL.PREAB, except that we set a different flag for 1RA.

bit 39, FL.IMAGE, is set to 1 when the image of a card just physically read is stored in the 16 CM words following the buffer, and reset to 0 when this card image is logically read and decoded.

In F5:
Bit 59, FL.OS, is set to 1 whenever we have just sent to the punch the image of a card that should be offset.
Bit 58, FL.PROS, is set to 1 whenever we have just set up the image of a card that should be offset.
Bit 59, FL.SU, is set to 1 when we have printed the message indicating that the rest of a print file will be format-suppressed.
Bit 58, FL.PRSU, is set to 1 when we recognize a type-in message calling for format suppression of a print file. Whenever it becomes possible thereafter to print the suppress message, we will do so and promote FL.PRSU to FL.SU.
Bit 57, FL.BCDO, is set to 1 when we have just sent to the punch the image of a BCD card.

Bit 56, FL.BCDN, is set to 1 whenever we have just set up the image of a BCD card.
Bit 57, FL.FMES, is set to 1 whenever we come to the end of a print file, as a signal that the next time printing is possible, the final message should be printed before terminating the file.
Bit 56, FL.CMEA, is set to 1 whenever we discover an unprintable character in a line just printed out of a file with a 48-character or 288-character chain, on a 512 printer. The next line printed will show which characters were unprintable, and at that time FL.CMEA will be promoted to FL.CMEB.
Bit 55, FL.CMEB, is set to 1 as described in the preceding paragraph, and causes the message UNPRINTABLE CHARACTERS IN PRECEDING LINE to be printed.
Bit 54, FL.8LL, is set to 1 when, in printing a file with a 512 printer, a format character causes 8-line spacing to be selected. This flag has to be set because a reset function may have to be given to the printer for unrelated reasons, and this function will have the effect of restoring 6-line spacing. FL.8LL will then cause us to re-select 8-line spacing before going on. The flag will be reset when a format character calls for 6-line spacing.

In F6:
Bit 41, FL.512, is always 1 if the file is being printed on a 512 printer.
Bit 40, FL.AUTO, is set to 1 when a format control character in a print file calls for automatic skipping from the bottom of each page to the top of the next.
Bit 39, FL.IMAGE, is set to 1 in a punch file, when we have set up a card image for punching, but have not yet sent it to the punch.
Bit 39, FL.PAGE, is set to 1 in a printer file (a) when we set up the final message at the end of a file, (b) when we switch the file to a different printer. When the line (the next line of the file in case b) comes to be printed, this flag forces a page skip beforehand. It is necessary because in case (a) the suppress flag FL.SU may be set, so that putting an appropriate format character in the line would have no effect; while in case (b) we want printing to begin a new page on the new printer, but do not want to disturb the format character in the next line.
Bit 38, FL.LONG, is set to 1 when we have terminated a print line image or a BCD card image not because we found a CM word that ended in a zero byte, but because we filled a card image (80 characters) or the print line storage space (310 characters) without finding such a terminator. The flag tells us, for punching, that if the first CM word for the next card image is zero, it should be skipped over, as it must have been the terminator for the preceding card rather than a word of information. For printing, the flag tells us that the first character of the next line, which is really just a continuation of this line, should not be used as a format character.
Bit 37, FL.SWCH, is set to 1 for a print or punch file whenever a type-in requesting a switch to a different unit is recognized. The switch may not be possible for some time.
Bit 36, FL.FLIP, is alternated between 0 and 1 to make us store successive punch card images in two areas alternately. This allows us to keep the images of the last card punched and its predecessor. A punch compare error is indicated by the hardware only for the predecessor, so that we have to re-punch both images.
Bit 36, FL.LINE, is set to 1 when we have begun setting up a print line image, but have not yet sent it to the printer.

Bits 48-53 of the 7th word of the FET for a print or punch file, which are found in F5, contain the disposition code copied from the FNT entry. For a print file, this can be 41B (501 printer), 42B (512 printer with 64 characters), 40B (either of the preceding), 43B (512 printer with 48 characters - never occurs in reality), or 44B (512 printer with 228 characters - never occurs in reality). For a punch file it can be 10B, for BCD, 12B, for normal binary, or 14B, for 80-column binary. FL.BIN and FL.PAB are equated to 2 and 4 respectively, to get cross-references when such a disposition code is tested (or in one case set).
Some of the bits of the FET status of the file, copied from the first word of the FET into CURFET4, are given FL symbols to provide cross-references:
Bit 0, FL.IDLE, is 1 when the file is not busy.
Bit 3, FL.EOF, is 1 when the file is at end-of-file (of course it is also 1, when the status is e.g. 13B).
Bit 4, FL.EOR, is 1 when the file is at end-of-record or end-of-file.
Bit 9, FL.EOI, is 1 when the file is at end-of-information.
Bit 2, FL.WRITE, is 1 when the file is being written. This is true only for a card read file being written on disk, and so its natural meaning is not interesting and is never referenced. But is is used as a fake in two ways: (a) for a print file to be backspaced, 1IU is called to another PP. It signals completion of its task by setting the FET status to 17B. (b) in a punch file, every end-of-file card is to be followed by a blank card. So after setting up the end-of-file card image the FET status is set to 0037B or 1037B (depending on whether it was previously 0033B or 1033B) and this indicates the next time the same file is worked on, that a blank card is to be punched before proceeding to the next file or to termination.
Bits 42-47 of the 7th word of the FET for a print or punch file, which is found in F6, contain the repeat count. This is normally 0; it is increased by 1 whenever a REP type-in for the file is recognized. When the printing or punching of the file has been completed, if the repeat count is not zero, it is reduced by 1, the file is rewound, and the process starts over. But if it is zero, the file is terminated.

In a card read file, bits 48 to 58 (which are found in G5) contain a count of the number of CM words of error messages that have accumulated during the reading of cards. Bits 24-38 contain the time limit from the job card.

Part of the 7th word of an FET remains unused:
For a card read file, bits 40-44.
For a punch file, bits 40-41, 54-55.
For a print file, bits 24-35.

The 9th through 14th words of an FET provide storage for 60 characters of display information. This begins with CR, CP, or LP, indicating the device type, followed by its EST ordinal as two octal digits. The 5th and 6th characters are blank. The 8th to 13th characters give the job name pertaining to the file, and the 14th character is blank (00B). If the 15th and 16th characters are 0000B, only the preceding information is displayed, on the J-display. If not, the 15th through 60th characters contain an operator message that will also appear on the J-display, and the whole line will also appear on the B-display provided that one of the four lines is available.

The 15th word of an FET contains the EST ordinal of the equipment, in binary, in bits 48-59, and normally contains 0 in bits 36-47. When 1IQ recognizes a type-in for the file, identified by the EST ordinal, it inserts a number indicating the kind of type-in, in bits 36-47. The rest of this word is not used except that for a backspace type-in (BS) the number of pages and the page format character are inserted by 1IQ in bits 24-35.

The 16th word of an FET is copied into cell LSTAT and cells DEEP through DEEP+3 when a cycle of work is started on the file, and those cells are copied back to the 16th word of the FET when the cycle is concluded. LSTAT contains a code number indicating the state of the hardware, so that when it changes from good to bad, or to a different kind of bad, the appropriate messages can be given once for the new bad status. When it changes from bad to good, the message if any is cleared, on the J and B displays.

Symbols

Here is a list of the symbols defined at the beginning of the assembly listing, apart from those already described in connection with the format of the FET:

ZERBUF, IR2, and IR3 are D.EST+5, D.BA+5, and D.BA+6, locations otherwise unused (except when 2TS is called), into which D.PPIRB+4, D.PPIRB+3, and D.PPIRB+2 are copied at initialization, leaving D.PPIRB through D.PPIRB+4 free to be used as next explained.

CURFET = D.PPIRB, and CURFET through CURFET+4 (CURFET+3 and CURFET+4 are called CURFET3 and CURFET4 for cross-reference convenience) hold the first word of the FET of the current file.

IN = D.FIRST and OUT = D.FIRST+1. The 8 cells D.FIRST through D.LIMIT+1 are not, in general, used normally. Instead , a subroutine called FIRST is provided and constantly updated so that

```
RJM FIRST = LDCA D.FIRST
RJM FIRST;ADD IN = LDCA D.IN
RJM FIRST;ADD OUT = LDCA D.OUT
RJM FIRST;ADD LIMIT,TYPE = LDCA D.LIMIT
```

Since a buffer is never more than 2000B word long, the "net" IN and OUT pointers can be held in single bytes, which simplifies some pointer arithmetic. It also frees the six bytes D.IN through D.LIMIT+1 for use as next explained.

LSTAT, DEEP, DEEP1, DEEP2, and DEEP3 are names for D.IN through D.LIMIT. LSTAT contains one of the statuses GOOD, UNREADY, REJFNC, XMSNPE, and WROGER, for the explanation of which see a few paragraphs below. They indicate approximately the status of the device for the current file.

DEEP through DEEP3 are used for saving and restoring the return addresses of subroutines that call LPC, the file-switching subroutine that has the effect of an interrupt.

CP30 through CP151 are equated to 30B through 151B, and are used to provide cross-references whenever words in the control point area are referenced. E.G., whenever control point +77B is referenced by the program, symbol CP77 will appear in the assembly listing. TR, FV and SX name the three cells after D.PPONE, in which constants 3, 5, and 6 are maintained.
WC names a working storage location in which word counts are held. PS, PS1, PS2, FETAD, and TYPE name the five cells D.BA through D.BA+4. PS contains the number of the FET of the file currently being worked on, and FETAD contains 16 times this number, which gives the address of the start of the FET. TYPE contains 0 if a punch file is being worked on, 1 if a print file, and 2 if a card read file. PS1 and PS2 are only occasionally used. I6789, I789, and I79, equal to 17B, 7, and 5 respectively, are used to provide references when dealing with 6-7-8-9, 7-8-9, or 7-9 punches in column 1 of a binary card image. NULLCH, equal to 0, is used as the channel number in every instruction involving a channel.
CMWORD and PPWORD, equal to 60 and 12, i.e. the number of bits in the two kind of words, are found only in the combination CMWORD/PPWORD, i.e. 5, used whenever the significance of 5 in the program is that 5 bytes make a CM word. INITA and INITB, equal to 3 and 7, are initial FET statuses that are found or inserted in CURFET4. INITB, if it occurs for a type of file, represents an initial phase just after INITA and before the real reading or writing begins. BCDCDLG and BINCDLG, equal to 8 and 16, are used whenever an instruction uses a constant having something to do with the number of CM words in a BCD or binary (respectively) card image.

HS.MEM, HS.NFEED, HS.XFEED, HS.TRAY, HS.CMPER, HS.BUSY and HS.READY are used as marks for testing the 12-bit status byte read from a card reader, punch or printer. HS.BUSY and HS.READY apply to all three device types; HS.CMPER (for compare error) to readers and punches; HS.TRAY refers to empty tray status in card readers; HS.XFEED refers to fail-to-feed status in card readers; and HS.NFEED refers to fail-to-feed status in card punches. S6.REJ, S6.IREJ, and S6,XPE are used as masks for testing the 12-bit status byte read from a 6681. S6.REJ refers to reject (either internal or external) status; S6.IREJ to internal reject status; and S6.XPE to 6681 transmission parity error status. HS.MEM refers only to 512 printers.

GOOD, UNREADY, REJFNC, XMSNPE, and WROGER are names given to the numbers 1 through 5 when stored in cell LSTAT and transferred between the cell and bits 48-59 of the 16th word of an FET. XMSNPE means that the last time a function was issued to the unit, a 6681 transmission parity error resulted. REJFNC means that the last time a function was issued to the unit, it was rejected. UNREADY means that the last time the unit status was found to be not busy, it was not ready, but not for the reason of XMSNPE or REJFNC. GOOD means that the last time the status was found to be not busy, it was ready. Status WROGER has nothing to do with hardware, but is stored in LSTAT for a print file from which a line is now being displayed on the console scopes. The line will cease to be displayed, and printing will resume, after an OK type-in by the operator.

B.MOVBF through B.LPEOR are symbols for the numbers identifying the various overlays. Only numbers 1 through 39 are allowed. The directory to the corresponding overlays is found at location INDEX, at the beginning of program 1IS, just before the listing of the first overlay.

MS.CFRA through MS.PRES are message numbers, used in calls to subroutine MAYMES. Actually the number of a message can range from 1 to 37B; if it is a B-display message, this is the number used in the calling sequence, while if it is a system dayfile message, 40B is added to the number to give the constant used in calling sequence. The corresponding messages are found in overlay B.MSG, and the directory to them is at MSDIC, at the end of the same overlay, in ascending order of message number modulo 40B.

RS.MOD through RS.END are message numbers, used in calls to subroutines UMES, which adds the corresponding message to the pre-OUTPUT file corresponding to the card read file now being processed. The messages are found in overlay B.UMES, and the directory to them is at UMES in that overlay, in ascending order of message number.

PC.DIS and PC.SKIP name the numbers to be added to the count of lines printed from a file when, respectively, a PM-line in the file causes printing to be suspended while the line is displayed, until the operator types OK; and when a line format character causes a considerable paper movement. The values now assigned are 1000 and 5. 1000 is the number of lines a printer should print in a minute, so that this value means that a PM-line is charged to the user on the assumption that the printer will be held up for a minute, while the operator does whatever the message calls for. PC.SKIP, the equivalent of 5 lines, is charged to the print file for functions 3 and 4 (advance to last line and page eject), 12B through 6B (select one of channels 2 through 6 for post print spacing), and 22B through 26B (select one of channels 2 through 6 for preprint spacing). These charges can be altered or eliminated by varying the definitions of these two symbols.

AL.CR, AL.CP, and AL.LP refer to card readers, punches, and printers respectively. After a card image has been read from a card reader, the next card image cannot be read until at least 50 milliseconds have passed. So immediately after reading a card image, 1IR adds AL.CR, which is equated to 33, to the current value of the millisecond clock, and stores the result as an alarm for this file. Then nothing whatever is to be done about this file until the millisecond clock has passed the alarm value. This avoids wasting the time of 1IR and the monitor on the file until 17 msec. before the earliest time at which the next card image could be read; this 17 milliseconds is plenty of time for housekeeping before that reading. In fact it may be too much, and it might be advantageous to increase AL.CR to some number closer to 50, say 40 or 45. Correspondingly, the minimum time between print line transmissions is 60 msec., and AL.LP is set accordingly to 40, though 50 might be better. The minimum time between punched card image transmission is 240 msec., and AL.CP is set accordingly to 200, though 220 or 230 might be better.

FET0 through FET15 are equated to integers 0 through 15, and are used in the program so that whenever the nth+1 word of an FET is read from or written to central memory, symbol FETn occurs in the listing and the cross-reference table.

BUFLG is the number of central memory words allotted to each buffer in the field length. This must be 400B or some higher integral power of 2. It must have the same value in programs 1IR and 1IQ. Note that the I/O buffer is actually shorter than this value by 20B for a card read file, or 40B for a print or punch file. The extra space, which follows the I/O buffer, is used to store the image of the last card read, or the images of the last two cards punched, or the image of the next line to be printed. LOGBUF is defined so as to be the logarithm to base 2 of BUFLG; it is used in shift instructions that have the effect of dividing or multiplying by BUFLG, in calculating the address of a buffer from its number or vice-versa. If k is the number in cell ZERBUF, buffer number n begins at RA+k+n*2**LOGBUF.

Narrative

1IR is loaded at 1000B, and jumps from there to PRESET in an area that will afterwards be overlaid by print line and card images. 1000B to 1777B is effectively vacant, and is used for loading overlays from 1IS, or for calling 2TJ through R.OVL. The stable part of 1IR begins at DROP, which is used to drop the PP, first zeroing bits 48-59 of control point +76B in order to inform 1IQ that 1IR is no longer active at the control point.

1. At PRESET, constants D.PPONE, TR, FV and SX are set. Then 1IR reads the input register, copies its last three bytes to IR2, IR3 and ZERBUF, sets the CP address, inserts the CP number into the 1IU call CALL1IU, reads the CP status word, sets D.RA and D.FL, and checks the error flag. If non-zero, it goes to DROP (see above); otherwise it reads the seconds clock and sets FORALARM and FNTALARM as different from it as possible, so that B.FORAG and B.FNT will be done the first time subroutine LPC is called. (They are the overlays that look for read and output files respectively.) Then it goes to LPCK, in subroutine LPC. Note that hereafter, D.PPIRB through D.PPIRB+4 are used as CURFET through CURFET4.

2. Most of 1IR, beginning at START, can be considered as one main loop, in which the program periodically returns to START. In fact, it can be considered as one separate loop for every active file. At many points in the loop, usually when for hardware reasons nothing can be done for the moment, subroutine LPC is called. This switches the program, if possible, over to the main loop for another file. The effect is almost as if each active file had its own PP looking after it. But as these imaginary PP's have to share a certain amount of variable memory, a lot of safeguarding has to be done. Most of it is done by subroutine LPC. But no overlay can call LPC, and no subroutine can call LPC without special precautions. No overlay, because by the time LPC switched back to the file that overlay had been concerned with, most likely some quite different overlay would have been loaded into 1000B-1777B. (LPC could have rigged to note what overlay(s) were in 1000B-1777B whenever attention was taken from one file, and to restore the same ones whenever attention next returned to that file. But as LPC is probably called several times for each line printed by any one printer, this would cause a tremendous amount of overlay - reloading

if several printers were busy. To keep LPC as quick as possible, the situation was left simple.) LPC cannot, in general, be called in the middle of a subroutine because the return address may have been changed in connection with another file before attention returns to the former file. However, subroutines READY and FILWT do call LPC; just before the call they save their return address, in effect, in the 16th word of the FET, and just after the return from LPC they restore it from the same place. In this they are imitating LPC itself. The principle is that every subroutine, including LPC, during which 1IR may switch from one file to another by altering PS, must have a separate return-address storage for every file.

3. If one imagines there being a different PP for each file 1IR is currently working on, each PP would contain the main loop of 1IR, without of course the calls to LPC, and without LPC itself. Also missing would be the overlays B.FNT and B.FORAG, which are logically part of LPC, and are executed at longer intervals to find new read or output files.

4. It is most practical to begin the narrative of the main loop by describing LPC, B.FNT, and B.FORAG. Observe that after the initial set-up at PRESET, the program goes not directly to START but to LPCK, in order to execute LPC before the first trip to START.

Subroutine LPC

5. This can be entered in two ways, either by RJM LPC or by LJM LPCK. If by RJM LPC, a file is being dealt with and must be put away before switching to another one. First, its alarm may or may not be set to the current value of the millisecond clock. Explanation of this will be left until the use of the alarm is described below. Next, copy the return address at LPC into DEST2; then copy D.FNT through DEST4 into the 6th, 7th, and 8th words of the FET. Then copy LSTAT and DEEP through DEEP+3 into the 16th word of the FET. When an exit is made from LPC, at LPCE, LPC calls subroutine PRE to fetch back the 6th, 7th, 8th, and 16th words of the FET that PS then points to, and then take the relevant return address out of DEST2 and branch to it.

6. This brings us to LPCK, to which we come directly without the above saving when the direct entry LJM LPCK is taken rather than RJM LPC. The LPCK entry is taken at five points in the program:

1. Immediately after PRESET, when 1IR has just been loaded and obviously there is no current file.

2. In overlay B.TER, when the current print or punch file has just been terminated; nothing is to be saved in its former FET area, which has just been zeroed, and its alarm is meaningless now.

3. In overlay B.FINIS, when the current print or punch file has been abandoned but left rewound in the output stack. Same as 2. above.

4. In overlay B.RTER, when a card read file has just been complete; analogous to 2. above.

5. In overlay B.JOB, when the job card of a card read file has just been processed. What the program between LPC and LPCK would have stored in the FET has been already stored by B.JOB, and as this overlay can only be called immediately after the card next after the job card has been read, the alarm will already have been set ahead by subroutine RCD.

7. At LPCK, the seconds clocks is checked to see if it indicates a different 4-second interval from FNTALARM. If not, go on to LPCF, but if so, call overlay B.FNT, from which we shall return to LPCL, read the clock again, and go on to LPCF.

8. As LPC is called many times per second, this means overlay B.FNT will be called about once every four seconds, to look for print and punch files. This looks perhaps insufficiently frequent. However, note that FNTALARM is set to a value 2048 seconds before or after the seconds clock when 1IR is first loaded so that B.FNT will be called on the first trip through LPC: and set 512 seconds ahead (in the overlay B.TER) whenever an output file is terminated. So the only time the 4-second interval can cause a delay of up to 4 seconds is when at least one printer or punch has been idle for a time, and a new file is put in the output stack. But if such idleness occurs when the system is in full swing, the system is apparently not printer-bound (or punch-bound as the case may be) and the overall through-put of the system is not reduced by the 4-second delay.

9. Whether or not overlay B.FNT was called, we come to LPCF to see whether the seconds clock indicates a different 4-second period from FORALARM. If not, we go to LPCG, but if so, call overlay B.FORAG, from which we shall return to LPCH; read the clock again, and go on to LPCG. This means that B.FORAG will be called about every 4 seconds, as well as the first time LPC is called. B.FORAG need not be called oftener as it is not used for initializing a file that follows its predecessor in the same card reader without a halt.

10. At LPCG we begin to decide which file 1IR should turn its attention to now. This depends on the 19 cells at ALARM+1 through ALARM+19, one for every possible FET except those for film, plot, and hard-copy files, which do not concern 1IR. These alarms are always kept by 1IR in its own PP memory. In turn, each one that corresponds to an active FET is compared with the current value of the millisecond clock, and in principle, the alarm that exceeds the clock by the greatest number indicates the FET to be chosen. In case of a tie, the lowest-numbered FET is preferred, but as will be seen, this does not cause a hang-up on that FET.

11. A difficulty arises here because the clock and the alarms have only 12 bits, and count so that 4095+1=0. So if clock minus alarm gives a negative answer, 4096 is added to produce a positive answer, the number of milliseconds by which the clock exceeds the alarm; i.e. the number of msec. since the alarm went off if it has gone off. Now whenever an alarm is set it is set to the current time in msec., modulo 4096, plus some number between 0 and 480. If an alarm appears to be x msec. behind the clock, either it is really so, or it is 4096-x msec. ahead of the clock. (Neglecting

the possibility that the alarm can be x plus a multiple of 4096 msec. behind the clock.) If x is less than 3584, then either the clock is 0-3583 msec. ahead of the alarm, or the alarm is 512-4096 msec. ahead of the clock. The latter is impossible, as the alarm is never set so far ahead of the clock; so the former is assumed, i.e. that the alarm has indeed struck. If x is greater than 3583, then either the clock is 3584-4095 msec. ahead of the alarm, or the alarm is 1-512 msec. ahead of the clock. The latter could be true and the former is very unlikely (it is unlikely that the file has gone without attention for nearly 4 seconds) so it is assumed that the latter is true and that the alarm has not yet struck.

12. Now consider again the beginning of subroutine LPC. If we enter normally, by RJM LPC, we there set the alarm for the current file equal to the current value of the millisecond clock unless the alarm appears to be less than 512 msec. ahead of the clock, in which case it is assumed it was recently set ahead and has not yet struck. (An alarm is set AL.CP=200msec. ahead of the clock on punching a card, AL.LP=40 msec. ahead of the clock on printing a line, and AL.CR=33 msec. ahead of the clock on reading a card. Exceptionally, the alarm is set 480 msec. ahead while waiting for program 1IU in another PP to complete the backspacing of a file, as this usually takes several seconds, and there is no point in thinking about it more than twice a second.) Each time LPC is entered, we set the alarm for the file that has just been current equal to the clock unless it appears to be waiting for a fixed interval after reading, punching, or printing. This makes the alarm for that file appear less urgent. Hence if two files have equal alarms, the one with the lower-numbered FET will get attention from LPC first, but its alarm will thereby be set forward, i.e. it will be come less urgent, and on the next entry to LPC, the other of the two files will be attended to. If this were not done, then when the file in the lower-numbered FET was suffering an indefinite mechanical delay, the other of the two files would be indefinitely delayed with no real reason.

13. As LPC is called several times for every card read or punched, and every line printed, it is very unlikely that an alarm has to wait more than 500 msec. before being looked at. So the difficulty caused by the circularity of the clock is hardly a real one.

14. Beginning at LPCG, all relevant alarms are checked. From bits 18-23 of the input register the number of the highest-numbered printer FET is obtained; then FET's numbered 1 through this number include all the reader, punch, and printer FET's. Then look in succession at PSTAK+1 through PSTAK+n. For each cell in this range that is zero, the corresponding FET is not active, and there is no need to look at its alarm. For each cell that is non-zero, 1 is added to D.Z5, merely to show that there is at least one active FET; and then the corresponding alarm is compared with the millisecond clock. If the alarm appears to have struck, as explained above, the apparent time since it struck is compared with D.Z6. D.Z6 is initially zero, and thereafter we put into it the greatest-so-far value of clock minus alarm. Every time a new value is put in D.Z6, the corresponding FET number goes into D.Z7, which was also initialized as zero.

15. After looking at all the active alarms, we check D.Z5; if it is still 0, there are no active files, and we go to LPCO. There we pause, and if there is no error flag go to DROP: 1IR will be recalled by 1IQ within 2 seconds at most. If there is an error flag, we go to DROPQ to ask monitor to set the error flag to F.ERPP; this will cause 1IQ to give up and JANUS to drop. Either the error flag was already F.ERRP, or it was something else and 1IR had to finish its current files, and 1IQ had to keep reappearing to service 1IR. Now that 1IR is idle, everything can drop.

    If D.Z5 is non-zero but D.Z7 is zero, all the active files appear to be waiting for their alarms to strike, so we wait one msec. and then return to LPCK to check them again. If D.Z7 is non-zero, it holds the number of the FET that appears to have been waiting longest since its alarm struck. At LPCD, this number is put in PS, 16 times it is put in FETAD as the address of the first word of the FET, and PRE is called to prepare to look at this file.

16. As this is the only time we call subroutine PRE, it is described now. Bits 6-11 of IR2 contain the number of the highest-numbered card reader FET, bits 0-5 do the same for punches, and bits 6-11 of IR3 do the same for printers. So by comparing PS with these values we find which type of device, and set TYPE to 0 for punch, 1 for printer, or 2 for reader. TYPE, like PS, is set only in subroutine LPC. Overlays B.FNT and B.FORAG also set them for their own purposes, but they are logically inside LPC.

17. At PREA, we read the first word of the FET into CURFET through CURFET4, and read the FIRST, IN and OUT pointers. These three pointers enable us to set FIRSTA so that subroutine FIRST becomes equivalent to LDCA D.FIRST, and to set IN and OUT to the differences between the FIRST and the IN and OUT, respectively, pointers. The sixth through eighth words are copied into the 15 bytes between D.FNT and DEST4; and the 16th word into LSTAT through DEEP3. This is the converse of what was done on entering LPC. But on the entry, neither the first word nor the buffer pointers were copied back into the FET; as changes to them are always copied immediately to the FET, for the information of stack processor. Also now, without any analogy to the entry procedure, we copy the 15th word of the FET into D.Z5 through D.T1 (see PREA.) We shall check this word for an operator type-in if this is a print or punch file, as the unit might be hung mechanically and preventing 1IR from reaching, for this file, START, where type-ins are normally handled.

18. Now if TYPE contains 2, for a card reader file, we exit from PRE back to LPCE. Otherwise, at PREB, we check D.Z5, which contains bits 36-47 of the 15th word of the FET. If it contains 4 or 5, indicating a type-in of SW (try to switch the file to a different unit of the same type) or REW (abandon work on the file, rewind it, and return it to control point 0 for a later try; then off the punch or printer in the EST) we (for 5) go immediately to FINIS to obey the REW, or (for 4) set FL.SWCH to 1, then zero bits 36-47 of the 15th word of the FET, indicating acceptance of the type-in; then exit from PRE to LPCE. If D.Z6 contains anything else, merely exit to LPCE. FL.SWCH, if set, will cause subroutine SW to switch equipment if possible, the next time it is called from subroutine RES or subroutine READY.

19. On returning from PRE to LPCE, a branch is made to the return address taken from DEST2. This is the converse of what was done with the return address on entering LPC, unless PS contained 0.

Overlay B.FNT

20. This part of the program looks for files to be printed and punched, and does the necessary initialization for printing and punching them. On entering it, FNTALARM is set equal to the current value of the seconds clock, so that the overlay will not be called again until the clock has moved into a different 4-second period, or FNTALARM has been set 2048 seconds away from the clock because 1IR is freshly loaded or set 512 seconds ahead of the clock because a punch or print file has just been terminated.

21. Now from IR2 and IR3 we get the number of the first punch FET, -1, and the number of the last printer FET, and call subroutine SRCHFET to see if any punch or printer FET is free. If not, the return is with a negative number in the A register and we exit to LPCL, as we cannot possibly start a new print or punch file. Otherwise, we scan the EST and zero AVPCH+1 if any punch is on and unassigned, zero AV501+1 if any 501 printer (LP) is on and unassigned, and zero AV512+1 if any 512 printer (LQ) is on and unassigned. Then, if all three cells have remained = 1, we exit to LPCL as there is no printer or punch available for a new file.

22. If there is at least one FET and one unit free, we request the FNT pseudo-channel, and at FNT2 scan the FNT for files that are not locked out, are not assigned to a control point, and not associated with non-allocatable equipment, are type output or local, are not EXPORT/IMPORT or RESPOND, and have a disposition code between 10B and 17B (punch) or 40B and 47B (print). Each time we find such an FNT entry, we set TYPE, at CFT1, to 0 for a punch file or to 1, 2, or 3 for a file with disposition code 40B (501 or 512 printer), 41B (501 printer) or 42B (512 printer). We are looking for the file with the highest priority in each of these classes, so now we compare the priority of the file with FNTPRI,TYPE; if the new file's priority is not greater, we merely go to FNT3 to continue the scan. If it is greater, it is stored in FNTPRI,TYPE; the address of the FNT entry is stored in FNTAD,TYPE; and the disposition code is stored in FNTDIS,TYPE.

23. When the whole FNT has been scanned, we arrive at FNT3B. Using D.Z1 as a counter, we scan FNTAD through FNTAD+3, representing the four classes of wanted file. If a cell contains 0, there was no file of that class; otherwise it points to the FNT entry for the highest-priority file of the class. We secure whatever files are pointed to by putting the JANUS control point number in their FNT entries; then we can drop the FNT pseudo-channel.

24. At FNT3E we see if there was apparently a punch available; if not, pass on to FNTPR. If yes, call subroutine SRCHFET to look for a free punch FET and put its number in PS. If the return from SRCHFET is with the A register negative, there is no free punch FET, and we pass to FNTPR.

Finally, we pass to FNTPR if FNTAD contains 0, showing that no new punch-disposition file was found. Otherwise, with the FET number in PS, 0 in TYPE to point to FNTAD+0 and FNTDIS+0, and to indicate a punch rather than a print file, and with "CP" in the A register to specify the needed type of equipment, call subroutine FSET.

25. This is actually the only place FSET is called. it saves the equipment type at APF10+1 and then requests the equipment. If not granted, we return to FNTPR. If granted, save the EST ordinal at PSTAK,PS. Then set D.LIMIT to contain 40B, indicating that the last 32 words of the nominal buffer area are to be saved for card or print line images, and call overlay subroutine B.SETFT. (Note that D.LIMIT is normally used as DEEP3, part of the 16th word of the FET. But at the moment we are not processing any file in the ordinary way, so the byte can be used freely.) B.SETFT, if it finds no buffer available, requests one from 1IQ and exits with the A register negative, whereupon we drop the printer or punch, zero PSTAK,PS, and go to APF7G (see section 39 below.) Otherwise, when B.SETFT returns, a buffer has been claimed for this file, the FET pointers have been set up, and subroutine FIRST and pointer bytes IN and OUT have been set.

26. Then we go to APF9, read the EST entry into DEST0 through DEST4, from which it will be copied into the 8th word of the FET, and alter its 3rd byte (DEST2) from the 3rd and 4th channel numbers (if any) to address START, to which subroutine LPC will branch the first time it handles the file. We alter the fourth byte (DEST3) from the type-letters of the unit to the address of the FNT entry.

    Into D.FIRST through D.FIRST+4 we read the first word of the FNT entry, containing the file name. (Note that these 5 cells are normally used as IN, OUT, LSTAT, DEEP, and DEEP1. But at the moment we are not processing any file in the normal way, so they are available.) These 5 bytes will become the first word of the FET. We alter the right byte to 3, the initial file status, and make the same change to the status byte of the third word of the FNT entry. Then we complete the formatting of the first word of the FET by setting bits 12-17 (in D.FIRST+3) to 0.

27. Next, just before APF10, we zero D.FNT through D.FNT+9, in which we shall format the 6th and 7th words of the FET as follows:

```
VFD    60/0
VFD    1/FL.OS,5/0,6/a,6/0,1/FL.512,5/0,36/0
```

    where a is the disposition code of the file; FL.OS is 1 for only a punch file, and means the last card of the preceding file (a blank) will now be offset; and FL.512 is 1 for only a 512 printer.

    Then we copy the 1st, 6th, 7th, 8th, and 16th (initially =0) words of the FET into the FET itself.

28. The last part of the FET to be set up is the J-display area in the 9th through 14th words, and the word for receiving type-ins, the 15th. The 15th word merely contains the binary EST ordinal in bits 48-59, and 0 in the remainder. The 11th through 14th words will be left zero initially, but we now set up the 9th and 10th words in BUF through BUF+9. BUF already contains the equipment type (see APF10), and BUF+3 through BUF+7 contain

the file name and status. We convert the EST ordinal to octal display code in BUF+1 and BUF+2; then zero BUF+7 through BUF+9, and write the 10 bytes to the 9th and 10th words of the FET.

Finally, we zero FNTAD,TYPE and FNTPRI,TYPE, to show the main part of B.FNT that we have disposed of the file; call subroutine SALARM to initialize the alarm for the file, and return from FSET to FNTPR.

29. When we reach FNTPR, we have certainly disposed of the punch file, if any, so we call subroutine SRCHFET to find a free print FET. If none, the return is with the A register negative, and we go to APF7G (see section 30 below). Otherwise, the subroutine has put the FET number in PS and its starting address in FETAD. We know we can write in the first word of the FET with no harm, so we use this fact to copy FNTPRI through FNTPRI+4 into D.T0 through D.T4, merely for convenience in the following comparisons.

    Now:

    1. If D.T1, D.T2, and D.T3 all contain zero, we have no print files outstanding and go to APF7G to terminate. (There may be some waiting in the FNT, but on each pass through B.FNT we only try to start one punch file, one 501 file, and one 512 file.) If AV501+1 and AV512+1 show that no printer is available, we also go to APF7G.
    2. If AV501+1 and AP512+1 show that one of each type of printer is available, we go to FNTPRF. Then:
        a. If D.T3 contains the highest or only priority, we go to FNTPRE. (Assign the 512-disposition file to the 512 printer.)
        b. If D.T2 contains the highest or only priority, we go to FNTPRB. (Assign the 501-disposition file to the 501 printer.)
        c. If D.T1 contains the highest priority (501/512-disposition file) we have a choice. If D.T2 contains the next-highest priority, we go to FNTPRJ to assign this file to the 512 printer, leaving the 501 printer free to handle the 501-disposition file later. Otherwise, we go to FNTPRK to assign the 501/512-disposition file to the 501 printer.
    3. If only one type of printer is available, we compare the priority of the file with specific disposition to it with the priority of the 501/512-disposition file (if any, in both cases), and send the winner to the printer. This means going to FNTPRE, FNTPRB, FNTPRK, or FNTPRJ.

    At FNTPRK, we set TYPE to 1, to indicate the 501/512-disposition file. At FNTPRB, set TYPE to 2, to indicate the 501-disposition file. In either case, we then set AV501+1 to 1, indicating that the printer will no longer be available the next time we come to FNTPR; then put "LP" in the A register to specify a 501 printer and go to FNTPRC, to call FSET.

    At FNTPRJ, we set TYPE to 1, to indicate the 501/512-disposition file. At FNTPRE, set TYPE to 3, to indicate the 512-disposition file. In either case, we then set AV512+1 to 1, to indicate that the 512 printer will no longer be available the next time we come to FNTPR. Then put "LQ" in the A register, to specify a 512 printer, and go to FNTPRC, to call FSET.

So in all we may go through FSET three times; the first time coming down from FNT3E, to deal with a punch file, and then twice via FNTPRB, FNTPRE, FNTPRJ, or FNTPRK, to send one file to a 501 printer and another to a 512 printer. This may not account for all the printers and punches that may be available for waiting files; if not, 4 seconds will elapse before B.FNT is called again. However, whenever a print or punch file is terminated, B.FNT is called immediately; so once all the printers and punches are busy, none of them is going to remain idle for 4 seconds if there are files waiting.

30. We come to APF7G when B.FNT has accomplished its agenda, or cannot complete it. We initially tried to find one punch file, one 501/512 file, one 501 file, and one 512 file. For each one found, the address of its FNT entry was stored in FNTAD+0, +1, +2 or +3, and the JANUS control point number was inserted in its FNT entry. As each one was assigned to an output unit and initialized, FNTAD,TYPE was zeroed. So now we look through FNTAD through FNTAD+3. Each non-zero is the address of the FNT entry for a file that we could not dispose of after all; we release that FNT entry from our control point. Finally, we return to LPCL.

35. So the FET for a punch or printer file is initialized as follows:

```
VFD     42/jobname,18/INITA
VFD     12/0,12/0400B,36/b,60/b,60/b,60/b+BUFLG-40B
VFD     60/x
```

seventh word for punch:

```
VFD     1/FL.PS,1/FL.PROS,1/FL.BCDO,1/FL.BCDN,2/x,6/d,
        6/r,2/x,1/FL.IMAGE,1/FL.LONG,1/FL.SWCH,1/FL.FLIP,
        12/e, 24/f
```

seventh word for printer:

```
VFD     1/FL.SU,1/FL.PRSU,1/FL.FMES,1/FL.CMEA,1/FL.CMEB,1/FL.8LL,6/d,
        6/r,1/FL.512,1/FL.AUTO,1/FL.PAGE,1/FL.LONG,1/FL.SWCH,1/FL.LINE,
        12/x,24/f
```

eighth word:

```
VFD     12/u,6/cb,6/ca,12/START,12/v,12/w
```

ninth to sixteenth words:

```
VFD     12/OHLP,12/qa,12/5555B,42/jobname,42/0
        (or 12/OHCP etc. or 12/OHLQ etc.)
VFD     60/0,60/0,60/0,60/0
VFD     12/qb,48/0
VFD     12/g,48/h
```

where:

b is the address of the start of the buffer; the names beginning with FL have been used illegitimately here, but are convenient to indicate the positions of the flag bits; all are preset to 0 except FL.OS, which is initially 1 and FL.512, which is initially 1 for a 512 printer file.

x has the value 0, and is used to indicate a field that is currently unused but is here preset to 0;
d is the disposition code of the file;
r is the repeat count, initially 0;
e is the card number within a normal-format binary record, initially 0;
f is the count of cards punched or lines printed, initially 0;
u is the control point number (unnecessary);
cb is the second channel number from the EST entry;
ca is the first channel number from the EST entry;
v is the address of the FNT entry for the file;
w is the equipment connect code;
qa is the EST ordinal of the equipment translated into two display code characters representing octal digits;
qb is the same ordinal as a binary integer;
g is what will be read to LSTAT, and is initially 0;
h is what will be read to DEEP through DEEP+3 and is not preset.

Overlay B.FORAG

38. This part of the program periodically checks all currently unused card readers to see if jobs are ready to be read in through them, and does the necessary initialization if so. On entering, the seconds clock is read and FORALARM set equal to it, so that the overlay will not be called again until the clock has moved into a different 4-second period, or FORALARM has been set 2048 seconds away from the clock because 1IR is freshly loaded. Next TYPE is set to 2, indicating that a card reader file is being dealt with, in case any subroutines common to all types are called. Overlay subroutine B.SKFNT is called to find a free FNT slot. If the return is with non-zero in the A register, there was no vacant space in the FNT for a new input file so overlay B.FORAG could do nothing in any case and branches back to LPCH. If the exit is with 0 in the A register, a vacant FNT slot has been found and filled with a copy of the model entry at MESS in overlay B.SKFNT, into which our control point number has been saved at PS1.

39. Next, the EST pointers are picked up and the EST starting address saved at DEEP. Now scan the EST for a card reader that is on and unassigned to any control point. When one is found, control goes to FORB. If there is none, we come to FORAGEY, where the FNT slot B.SKFNT had reserved is released, by zeroing it, and we then exit from the overlay to LPCH.

40. At FORB, the EST ordinal of the unassigned card reader is calculated and saved in PS2; then a specific request for it is constructed, and sent to monitor. If the request is not granted, scanning of the EST continues at FORC. If it is granted, a free FET must now be found. FET's numbered 1 through p are for card readers, where p is in bits 6-11 of IR2. So PSTAK+1 through PSTAK+p are scanned for a zero cell that indicates that the corresponding FET is free. If none is free, we drop the card reader, at FALCON, and go to FORAGEY to drop the FNT and exit from the overlay.

41. If a free FET is found, we go to EDSEL with its number in PS, construct its starting address, and store this in FETAD. LSTAT is set to contain the status code UNREADY; when we check the card reader status in a moment, the most probable status will be unreadiness. To prevent this from giving a pointless message, we anticipate this status in LSTAT. Next we put the EST ordinal twice in the FET: as a binary integer in bits 48-59 of the 15th word, so that program 1IQ can find the FET for this card reader when there is a type-in for it; and as a two-digit octal number in display code, preceded by the letters CR and followed by 10 blanks and a zero byte, in the 9th word and bits 24-59 of the 10th word. This enables DSD to show the card reader number, followed by a blank name, in the J-display between now and the time the new job name is read from a job card, or the card reader is dropped. It also enables 1IR to output a message concerning the card reader on the B-display, if this is necessary before the job card has been successfully read. Then LSTAT, which was initialized as UNREADY (=2), and DEEP through DEEP3 are copied into the 16th word of the FET.

42. The EST entry for the card reader is still in D.EST through D.EST+4 (DEST0 through DEST4). This will become the 8th word of the FET later at FOREA, when the FNT pointer and the program address START replace the 3rd and 4th channel numbers and the type letters "CR". Subroutine RES is called to reserve a channel and connect the reader; and subroutine STS, to read its status and return it in the A-register. If we have non-zero in the A-register on return from RES, however, the card reader connect code has been rejected and we go to FORECS to drop the reader and the FET and return to FORC to continue scanning the EST for card readers. Subroutine RES will already, in this case, have called subroutine RELCR to drop the channel and deselect the 6681 and card reader.

43. But if we return from RES with 0 in the A-register, and then from STS with the status, and if the status is ready and tray not empty, we go to FOREA. Or if the status (which subroutine STS also stored at location STAT) is tray empty or fail to feed, we call subroutine RELCR to release the channel, 6681, and reader, and then go to FORECS to drop the reader assignment and the FET and return to FORC to continue scanning the EST. If the status is not ready, but not because of tray empty or fail to feed, we call subroutine NRD to deal with what may be a bad card, and then go to FORECS; RELCR will have been called by NRD.

44. At FOREA, everything is apparently ready to read a card, but a buffer is still needed, and the rest of the FET must still be filled in. The FNT entry address is put in DEST3 (where it replaces the letters "CR" from the EST entry) and the address START in DEST2 (where it replaces the third and fourth channel numbers, if any; we assume that the card reader is attached to only one or two channels) to be the address to which subroutine LPC will exit the next time we turn our attention to this card reader file we are now initiating. This completes the modification of the EST entry into the eighth word of the FET, so we copy it into the FET. Next we call overlay subroutine B.SETFT, with 16 in D.LIMIT, to get a buffer and set up FET pointers

such that the last 16 words of the nominal buffer area are left for card image storage. (Note that D.LIMIT is normally used as DEEP1, but as we are not now processing a file in the normal way, the cell is available.) If B.SETFT cannot get a buffer, it requests one from 1IQ and returns with the A register negative; then we go to FOREL to release the channel and card reader, and thence to FORECS as if the card reader had not been ready in the first place. Otherwise, a dummy job name 9999 is put into the first word of the FET, with status INITA (=3). Then, at last, subroutine RCD is called to read the waiting card. (Note that RCD is entered with non-zero in the register; otherwise it would not physically read a card.) But it is not examined; RCD has transferred it to the hold area after the end of the buffer. The EST ordinal of the reader, which is still in PS2 is stored, at PSTAK,PS to indicate that the FET is now active. Then the 7th word of the FET is set to 0, except for bit 39 (FL.IMAGE) which, being 1, will indicate that there is a card in the hold already the next time RCD is called. The sixth word of the FET must be zero, and it is left so. Then return to LPCH. The next time subroutine LPC brings attention to this file, the second card will be read, and the fact that the FET status is INITA will cause overlay B.JOB to be called, to decode the job card.

47. Control comes to FORECS after some sort of non-success, when an FNT slot, an FET, and a card reader, but not a buffer have been secured. As the EST ordinal has not been put in PSTAK,PS, however, the FET has not been formally claimed and so is not formally dropped; but the whole FET area is zeroed out again. The card reader is dropped as well, and then a return to FORC is made to continue scanning the EST.

48. The summary of the initial format of the card reader FET is deferred until we describe overlay B.JOB, where the information from the job card is inserted in the FET.

The Main Loop of 1IR

49. The main loop of 1IR, beginning at START, can be described more or less apart from subroutine LPC and its attendant overlays B.FNT and B.FORAG. At several points, where nothing useful can be done for the moment on the current file, there are calls to LPC that in reality will send the PP to a completely different part of the main loop. But the next time the current file becomes current, the program will return to the point in the main loop at which the LPC call last took it away from the file. This is why it may be useful to think of the main loop as being several loops, one for each active FET, with the LPC calls acting as cross-links.

50. The current file is identified primarily by PS, the FET number (varying between 1 and some upper limit), and FETAD, containing 16 times this number, the starting address of the FET. At START, we read the 15th word of the FET, and see if bits 36-47 contain zero. If so, branch straight to STARTA, but if not they contain a number representing some operator type-in addressed to this file via its equipment EST ordinal, which is permanently in bits 48-59 of that word of the FET. The number is saved in D.Z6 and bits 36-47 of the 15th word of the FET zeroed, to show the type-in has

been accepted. If the type-in was for backspace, the parameters, taken from bits 24-35 of the same word, are still in D.Z3. If the number in D.Z6 is above the limit, a branch is made to STARTA anyway as an invalid number has somehow appeared. Otherwise, the branch is to one of the locations shown in table STARTB:

Type-In	Number	Branch to	Function
END	1	END	End the file (input or output)
SUP	2	SUP	Suppress format control (print)
REP	3	REP	Repeat the file (print or punch)
SW	4	SWC	Switch the file (print or punch) to a different unit as soon as one is free, and off the present unit in the EST.
REW	5	FINIS	Stop printing or punching the file; do not charge for the work already done on it; rewind and replace it in the output stack; off the present unit in the EST.
OK	6	OK	Stop displaying a PM-line in a print file and resume listing at the next line.
BS	7	BACK	Backspace a print file.

51. These functions will now be described, before resuming the main loop.

END

52. If TYPE contains 2, indicating a card read file, control goes to ENDCR. There subroutine UMES is called to add the message OP.DROPPED to the future OUTPUT file of the job. Then FL.PREAB is set to 1, indicating that the job is to be aborted immediately whenever it is brought to a control point for execution. Then continue at REOFL, as if a 6-7-8-9 card had just been read.

53. If TYPE does not contain 2, so that the action is printing or punching, subroutine FILWT is called with 0 in the A register. This waits until the FET shows not busy, i.e. no reading from disk going on, and does it in such a way that subroutine LPC can be called repeatedly while waiting, so as not to hold up the other files. Then check TYPE again. If it contains 0, indicating a punch file, overlay B.END is executed. This overlay begins be calling subroutine CLERMES, to set LSTAT to GOOD and clear any B-display message that may be in the FET. Then it calls subroutine WTFET to set the code and status field of the FET to 1033B, i.e. end-of-file and end-of-information. Then it calls subroutine overlay B.SETP with 16 in IN. This sets the FET pointers so that OUT=FIRST and IN=FIRST+16. Then it sets up a card image with 7-9 punches in column 1 (so that if the card is ever read by mistake in an input file, it will appear as a binary card of impossible format, and will cause the job to be pre-aborted) and a large visible E in the middle of the card. This card image is copied into the first 16 words of the buffer, so that it appears to constitute the last 16 words of information

in the file. Then the disposition code in bits 48-58 of the 7th word of the FET is set to 14B, for 80-column binary (what it was before no longer matters) and we branch to STARTA. The main program will now punch that signal card image, then see the end-of-file and end-of-information and punch a 7-8-9 card then a 6-7-8-9 card, and finally terminate the file.

54. But if TYPE contains neither 2 nor 0, control goes to ENDLP and calls overlay B.LEND. What must be done is print a message ****ENDED BY OPERATOR **** then print the dayfile at the end of the file, if it is of type output rather than local, and finally repeat or terminate the file. We have already waited for the file to be not busy. Now, in the overlay, overlay subroutine B.SETP is called to set the buffer pointers to empty. Then we set FL.LINE=1, and copy what begins at ENDMES into the line image area. The next time we switch to this file, control goes to LPRL and we print the message. The control word begins with 6031B rather than 4031B because in case of extended array printing the message will have to be translated from display code. (What follows at ENDLQ could not be included in the overlay because it involves the calling of subroutine FILWT, which calls LPC, which must never be called even indirectly from an overlay.) With 240B, the FET/FNT code for skip forward, in the A register, a call is made to subroutine FILWT, which waits till the file is not busy, inserts the code in the FET and the FNT entry for the file, and partly sets up a stack request at MYRQ through MYRQ+9. Then the FWA field of the stack request is replaced with 777777B, to ensure a skip forward to end-of-information. Then, with function code O.SKF in the A register, subroutine DORQ is called to complete the stack request and get it issued. This forward skip does not affect the buffer or the FET pointers, so when the skip is completed it will appear that the message in the buffer is the very last thing in the file; returning to normal procedures will cause it to be printed and then processing continues as if the file had normally printed through to the end. This is fine if the file is not of type output, and control goes to ENDQ to wait for completion (by calling FILWT with 0 in the A register) and then calls subroutine CIO and goes back to LPD99. CIO will do nothing because of the end-of-information status. But for an output type file the dayfile must always be printed, even after an END type-in. So a backspace of one logical record from the end of the file is performed, which should position us at the beginning of the dayfile. 40B, with which FILWT is entered, is the FET/FNT code for a simple backspace of one logical record. But what really controls the outcome of the manoeuvre is the function code O.SKB, with which DORQ is entered, and the logical record count of 1, which the FWA field of the stack request, in MYRQ+6 and MYRQ+7, is replaced between the calls to FILWT and DORQ. After this stack request has been completed, it will not have affected the buffer or the FET pointers, so the message will still be in the buffer and be positioned at the beginning of the dayfile. At ENDQ, then, this completion is awaited by calling FILWT with 0 in the A register, and then calling subroutine CIO to start reading the dayfile, and returning to LPD99.

SUP

55. If TYPE does not contain 1, for a print file, control goes straight back to STARTA as there is nothing to be done. For a print file, if either FL.SU or FL.PRSU is 1, the same happens. Otherwise, FL.PRSU is set to 1, so that the next time the main loop for this file finds it must begin constructing a new print line image, it will see the pre-suppress flag, reset it to 0, set FL.SU the suppress flag to 1, so that the format characters of all future lines will be ignored; then insert the message **** FORMAT CONTROL SUPPRESSED **** in the line image store and go ahead as if this message had been taken from the buffer, just before the remainder of the file. This is done in overlay B.SUP, which is described at the only point in the main loop where it is called.

REP

56. Overlay B.REP is immediately called. In the overlay, if TYPE contains 2, indicating a card read file, control goes to STARTA as the type-in has no meaning. Otherwise, 1 is added to the repeat count in bits 42-47 of the 7th word of the FET. This field is normally 0; 1 is subtracted from it whenever punching or printing the file is completed, and when this would yield a negative result the file is terminated. According as this field has just been made 1, 2, 3, 4, 5, 6 or greater, the character in bits 42-47 of the 10th word of the FET is altered to - = * / + (or). (When the repeat count is 0, this character is blank. It is the character after the seventh character of the file name, as it appears in the J-display and sometimes the B-display. The characters - = and * are chosen because in their scope display forms they have 1, 2, and 3 strokes. The rest of the characters in the set are arbitrary, but the repeat count is not likely to go above 3 in practice.)

SWC

57. If TYPE contains 2, indicating a card read file, go immediately to STARTA as the type-in is inapplicable; one cannot read the rest of a file on a different card reader if the first reader breaks down in the middle. Otherwise, set FL.SWCH to 1. This is a signal to subroutine SW, which is called whenever a printer or punch is found not ready or rejects a function, that the file should be transferred to another unit of the same type, if there is one available, and that the current unit should then be turned off in the EST.

FINIS

58. If TYPE contains 2, indicating a card read file, the REW type-in is treated as an END type-in and there is a branch to ENDCR. Otherwise control waits till the file is not busy, by calling subroutine FILWT with 0 in the A register; then calls overlay B.FINIS. The overlay

begins by calling overlay subroutine B.REW, which rewinds the file and leaves a copy of the FNT in MYFNT through MYFNT+14. Next the file is returned to the stack by altering the control point number in the first word of the FNT entry to 0, at the same time setting the priority to 7777B so that the file will restart punching or printing as soon as possible. Then overlay subroutine B.DROPBF is called, to release the buffer; we zero all 16 words of the FET; turn off the unit whose EST ordinal is in PSTAK,PS by setting bit 23 of its EST entry to 1; call monitor to release that unit from our control point; zero PSTAK,PS to show that the FET is no longer in use; set FNTALARM ahead by 512 seconds, so that overlay B.FNT will be called the next time subroutine LPC is entered, and branch to LPCK in that subroutine. Here the abnormal entry to it is taken because nothing need be saved back into the FET just cleared.

59. The net effect is to abandon the printing or punching of the file without giving any further message, and without putting an accounting message in the dayfile, leaving the file in a position to be re-processed as soon as possible, and turning off the unit so that the attempt to reprocess the file does not simply bring it back to the same unit again.

OK

60. If TYPE contains 2, indicating a card read file, control goes straight to STARTA as the type-in is inapplicable. Otherwise, subroutine CLERMES is called before returning to STARTA. The only time this will be a useful procedure is when the type-in is addressed to a print file that has suspended printing while displaying a PM-line on the scope. Whenever that file is the one to which PS and FETAD point, LSTAT will contain status WROGER (=5); subroutine CLERMES will clear out the message field in the FET and reset LSTAT to GOOD. The next time the main loop comes to the file, this status will allow printing to resume.

BACK

61. If TYPE does not contain 1, control goes straight to STARTA, as the BS type-in is applicable only to printers. Otherwise, the call to program 1IU is completed at CALL1IU, by inserting the FET number in bits 0-11 of the 1IU call, and copying into bits 12-23 of the 1IU call what was in bits 24-35 of the 15th word of the FET. Then a PP is requested for this call, with zero delay.

62. What was in bits 24-35 of the 15th word of the FET, when 7 was found, asking for backspace, in bits 36-47, is the page count and page format character 1IU to use. This is described more fully in the documentation of 1IQ (see subroutine CONSOLE) which transmits the type-in from DSD to 1IR, and the documentation of 1IU, which actually backspaces the file.

63. When 1IU has completed its backspacing, it will set the status field of the first word of the FET to 17B. This is the only time in JANUS that a print fiel has this status. It is repeatedly tested for, and after each unsuccessful test we set the alarm for this file 740msec.

ahead (probably there are several seconds to wait, and LPC should turn to this file as rarely as possible in the meantime) and call LPC. When status 17B appears, the FET status is reset to 13B and the return is to LPD99. Note that LPC stores the 6th, 7th, 8th, and 16th words of the FET every time it is called from this part of the program, but reads back to the PP these four words, plus the first word and the four pointers. So 1IU can use the first five words of the FET without risking a clash with 1IR.

Main Loop of 1IR Continued

64. Control comes to STARTA if there was no type-in message to deal with, or if it was a message (like REP) that could be obeyed without any detour in the main loop. Now one other preliminary common to all three types of file is performed, namely to consider moving to a buffer lower in memory. The low-order 12 bits of the card/line count are taken, increased by 1 and the result checked for divisibility by 200B for punch file, or by 1000B for read or print file. If not, branch to START7 where the file is actually worked on. But if so, shifting buffers is considered. The numbers 200B and 1000B, which are set by the three constants at INTER (and of course could be altered to any other three powers of 2) are designed so that buffer-moving will be considered for each file at intervals of about 30 seconds. One is added to the card/line count before masking it so that buffer moving will not be considered when the count is initially 0.

65. To consider shifting buffer, subroutine MAYMOV is called, which decides whether the following conditions are satisfied:

    1. Out buffer is not one of the first two. Storage will never deliberately be shrunk below the point of having one free buffer in the field length; ours and the free one make two, so there is never any point in shifting buffer assignment down from the second buffer.

    2. Out buffer is the highest-addressed one currently in use.

    3. There is at least one free buffer below ours. If the conditions are not satisfied, and exit is taken from MAYMOV with the A register non-zero, and a branch to START7. If they are satisfied, an exit is taken from MAYMOV with the A register zero; subroutine FETCH has been called and subroutine PUT has not subsequently been called, so that the interlock is favorable and the buffer assignment registers available. However, the business cannot be done yet, so subroutine PUT is called to drop the interlock.

    Then subroutine FILWT is called to wait until the FET shows not busy - obviously the buffer cannot be moved while disk reading or writing is going on. Because FILWT may call LPC, which must not be called from an overlay, the FILWT call must be in the permanent part of 1IR. After the call to FILWT, the overlay subroutine B.MOVBF is called. This subroutine will copy the contents of our buffer to a lower one if it is still possible, and adjust the pointers in PP memory and in the FET accordingly. The exit from the subroutine brings control to START7.

66. START7 is reached on finishing the preliminaries that are common to all three types of file. If TYPE contains 2, control goes to CRD to work on a card read file. Otherwise, there is a print or punch file. If LSTAT contains WROGER (=5) this file must be a print file that is displaying a PM line and waiting for an OK type-in. So control goes to LPD99 to call LPC and eventually returns to START. Otherwise, beginning at LPD1E the number of CM words remaining in the buffer is calculated, and stored at WC. Note that LIMIT-FIRST necessarily = BUFLG-40B. Then branch to LPR for a print file, or continue at PCD for a punch file. Just before branching, however, call subroutine CIO to refill the buffer if it contains fewer than 63 CM words, and if the FET status is read completed.

67. First of all, for a punch file, a check is made to see if the file status in the FET is INITA (=3); if not, go to PCDA. If so, this indicates that nothing has as yet been read from the file nor anything punched out, and the job name card must be set up. So verlay B.INIT is called. It begins by calling overlay subroutine B.FCIA, which is really just a table of character images, made into a separate overlay so that it can be used by both B.INIT and B.LINIT. Executing it merely causes the table to be copied into the high end of the overlay area. Then a card image is set up with all 1's in the top two and bottom two rows, and a blank area in the middle. However, columns 1 and 2 are left for the moment, and column 3 is set all blank. Then the seven letters of the file name are taken from CURFET through CURFET+3 and converted into visible punching, beginning in column 4. Each letter or digit is 10 columns wide, and there is one extra blank column between each pair of neighbors. The visible characters are only six rows high, 1-row through 6-row, so one character image, ten columns on a card, can be packedinto five PP memory cells. When the image is complete, control goes to FCI4, and calls subroutine WTFET to set the FET status to 53B, indicating that the file is no longer in initial condition. Finally it takes the last two characters of the job name, which are the crucial ones in identifying it (since they are assigned by the system), gets the corresponding Hollerith punches from table HOLLER (also in overlay B.FCIA) and puts them in columns 1 and 2 of the card image. This is to give a way out of possible confusion between letter O and and digit 0, or 5 and S, or 2 and Z (though the images are different for these pairs). Finally return to the permanent part of 1IR at PCDK; which is where we ordinarily arrive immediately after reading enough words for a card image from the buffer, setting up the card image, and calling subroutine SAVOUT to update the OUT pointer in the FET.

68. In the non-initial situation, control comes to PCDA instead of calling B.INIT. Then if FL.IMAGE is 1, control goes straight to PCDL as there is already a card image waiting to be punched, in one of the storage areas that follow the buffer. Otherwise, a new card image must be constructed from words in the buffer. If WC contains non-zero, the buffer is not empty, and control goes ton to PCDB to do this. Otherwise, the buffer being

empty, if the FET status is end of record/file overlay B.EOPCH is called to set up the corresponding image of a 7-8-9 or 6-7-8-9 card and then branch to PCDK or PCDKA. If the buffer is empty and the status is not end-of-record/file, it may be end-of-information; if so overlay B.MTER is called at MTER; or if not, control goes to LPD99 as there is nothing to do but wait until there is something in the buffer. (End-of-information without end-of-record may seem odd. When the FET status first becomes e-o-i, it also becomes, of course, e-o-r or e-o-f. We process the e-o-r or e-o-f in overlay B.EOPCH., and set the status to 0013B if not already e-o-i, or 1013B if e-o-i. Subroutine CIO will never read into the buffer while the status is e-o-i, so 1013B will remain unchanged until it is picked up, as noted above, and control goes to MTER, with the advantage of having already punched out the 7-8-9 and/or 6-7-8-9 cards in the normal way.)

69. Before discussing the treatment of normal information cards, overlay B.EOPCH will be described. This begins by calling subroutine CLR to zero IMAGE through IMAGE+79, where the proper card image will be initially constructed. Then it zeros PCT so that if a record that will be followed by a normal binary record in the same file is being terminated, the numbering of cards in the new record will begin at 1. Now, if FL.WRITE is 1, a fake write status (actually 0037B or 1037B) exists, which must have been set on the preceding entry to the overlay, just after constructing a 6-7-8-9 card image. Such an end-of-file card has to be followed by an offset blank card. The image of a blank card is already in IMAGE through IMAGE+79, so now, at EOPE, the FET status is set to 0013B or 1013B; then, at EOPB, FL.PROS is set to 1 (indicating that the image that will next be sent to the punch is for an offset card) and we return to the permanent part of 1IR at PCDKA. PCDKA is also reached when the image of a normal information card has just been set up in the permanent part of 1IR, and FL.PROS set to 0, as such a card is not to be offset.

70. If FL.WRITE is 0. there is a branch to EOPA. If the status is end-of-record, go to EOPC. Otherwise, check FL.OS. If this is 1, the last card punched must have been a 7-8-9 card, and control can proceed with the 6-7-8-9 card, by storing I6789 (=17B) in the first column, at IMAGE, setting the FET status to 0037B or 1037B (the fake write status is explained above), and going to the permanent part of 1IR at PCDK. As the 6-7-8-9 card is not to be offset, it goes to PCDK rather than PCDKA, so that FL.PROS will be set to 0 before control arrives at PCDKA.

71. But if end-of-file, non-fake-write status exists in the FET and FL.OS is 0, we go to EOPD. An end-of-file that was not immediately preceded on disk by an end-of-record has been reached, but an offset 7-8-9 card should be inserted before the 6-7-8-9 card. I789 (=7) is stored in column 1 of the image, at IMAGE, and a branch back to EOPB is made, to set FL.PROS and return to PCDKA. The next time control comes through the main loop for this file, it will enter this overlay, and FL.OS will be set (FL.PROS is promoted to FL.OS immediately after punching a card.) Thus it will set up a 6-7-8-9 card the next time it comes to EOPA for the file, rather than branching to EOPD.

72. At EOPC there is an end-of-record status in the FET. Bits 14-17 of the first word of the FET (bits 2-5 of CURFET3) represent the level number, which must be translated into two octal digits and represented as two Hollerith punches in columns 2-3 of the card image. So bit 5 of CURFET3 is passed through subroutine EOPT, which converts it to the Hollerith column image for 0 or 1, which is stored in IMAGE+1, then bits 2-4 of CURFET3 are sent through EOPT, and the returned image of something between 0 and 7 stored in IMAGE+2. Finally I789 (=7) is stored in column 1, at IMAGE, and control goes to EOPE. EOPE was reached above after the preparation of a blank card image, when 0037B or 1037B status was found, calling for an offset blank following a 6-7-8-9 card; the end-of-record card whose image has just been set up is treated the same in all respects except for what is actually the image.

73. Return is made to the program just below PCDA. If the buffer was not empty, one or more information cards had to be punched before doing anything exceptional, and control went to PCDB. Now at PCDB the disposition code is checked, and we go to PAB for 80-column binary (14B, i.e. 10B+FL.PAB), BIN for normal binary (12B, i.e. 10B+FL.BIN), or HOLP for BCD (10B).

74. PAB is the simplest case. WC is set to contain 16, the number of CM words of information the card can hold, and subroutine WFR is entered with IMAGE in the A register; this is the starting address into which the words are to be transferred. WFR will move 16 CM words, i.e. 80 PP words, from the buffer to IMAGE ff. VFD 12/PCDE in the calling sequence to WFR is the address to which WFR is to branch if it exhausts the buffer and finds end of record/file before it has moved 16 CM words. Thus control goes to PCDE whether or not this happens. This means that when an 80 - column binary record does not have a length divisible by 16 CM words, the last 1 to 15 words are punched in columns 1 ff of the last card, and the remaining columns are merely left blank.

75. At BIN a standard binary card is punched. This is a little more complicated. WC is set to contain 15, the number of CM words of information it can contain, IMAGE+2 goes into the A register (information on a standard binary card begins in column 3), and subroutine WFR is called. VFD 12/BINA in the calling sequence means that the return is from WFR to BINA whether WFR finds 15 CM words (i.e. 75 PP words) in the buffer and transfers them to IMAGE+2 through IMAGE+76, or whether it finds end-of-record/file status after fewer words in the buffer, which it has put in IMAGE+2 ff. In the dozen or so instructions beginning at BINA, a checksum of the contents of columns 3 through 77 is formed and stored in column 2 of the card image. The checksum is formed by totalling the 75 bytes as if in a 12-bit register with end-around carry, and taking the 1's complement of the result. Subroutine WFR has stored in D.Z5 the number of words it actually moved into the card image, whether 15 or some smaller number, and this is now used to set up column 1 of the card image, which contains a word count in the upper half and 7-9 punches in the lower half. Then control passes to PCDE as it did after preparing an 80-column card image.

76. At HOLP a BCD card is punched. From HOLP down to HOLPB is concerned with transferring the first CM word to the card image, and from HOLPB to just before HOLP52 is concerned with transferring the second and following words. First WC is set to 1, as subroutine WFR will be called for each single word to be transferred. WC need not be set again below, as WFR does not alter it. D.Z1 is set to contain IMAGE, the address into which we start moving output words; observe that this is a pointer for HOLP, not for WFR. D.Z4, in which to count the number of CM words transferred to the card image, is zeroed. The A register is set to contain IMAGE and subroutine WFR is called in to read one CM word from the buffer to IMAGE through IMAGE+4. VFD 12/* in the calling sequence indicates that this return from WFR is impossible. WFR is only asked to get 1 word, and it is known that the buffer is not empty.

77. Now if FL.LONG is 1, the card before the current one in the record ended not because a word with a final zero byte was found, but because 8 CM words of information were taken from the buffer without finding such a word; this filled the preceding card. If this flag is 0, a jump is made straight down to HOLP2; if 1, it is reset to 0 and then a check is made to see if the word just brought from the buffer is entirely zero. If not, control goes to HOLP2. But if so, this word is merely a terminator to the preceding card, which containd 40 bytes of information; this word in D.T0 through D.T4 is not to be translated, as it normally would be, into a blank BCD card. So the word of zeros is thrown away irrecoverably, by calling subroutine SAVOUT to set the FET OUT pointer past it (subroutine WFR advances the PP pointer). This brings control to HOLPB, where we load the A register from D.Z1 and call subroutine WFR to move the next signle word from the buffer to the next unused 5 bytes after IMAGE. VFD 12/HOLP52 in the calling sequence means that if the buffer turns out to be empty, and the status is end-of-record/file, WFR will branch to HOLP52 instead of exiting normally. There, we will find D.Z4 still containing zero, and so return to LPD99 without having punched a card. The record ended with a card containing 40 bytes of information, which was punched on the last trip through the main loop for this file, and on this trip nothing has been done but see the zero word that terminated that card image. On the next trip, the end of record card will be punched. If WFR does not branch to HOLP52 for this reason, control arrives at HOLP2 as if the first word read from the buffer had not been all zero. If it was zero, it has now been completely discarded.

78. At HOLP2, if the last byte of the word just fetched by WFR is zero, this is the last word in the card image and control goes to PCDE via HOLP5. Otherwise, add 5 to the destination pointer in D.Z1 and 1 to our count of CM words processed in D.Z4. If this reaches 8, the card image has been filled up; otherwise control goes back to HOLPB to continue it. Note that it did not come through HOLPB, in the ordinary case, for the first word on the card image. Now, at HOLPB, subroutine WFR is called to move the next word from the buffer to the next unused 5 cells after IMAGE, or to branch to HOLP52 if the buffer is empty and the status is end-of-record/file. Arriving at HOLP52 in this way, we will not find zero in D.Z4, as at least one word has been moved into the card image,

so we will punch the card, etc. in the ordinary way. This is a case of the record ending with a card image containing exactly 5, 10, 15 .... or 35 bytes of information, without a zero word to terminate it. If we do not get to HOLP52 this way, we return to HOLP2 at the top of this paragraph.

79. If D.Z4 reaches 8 before a word ending in zero byte is found, the card image has been filled and must be punched out. The next word in the record might, after all, be the terminator of this card image; to prevent it from being punched out as a blank card on the next trip through the main loop for this file, FL.LONG is set to 1 before going to HOLP5. The effect of this flag has already been explained.

80. At HOLP5, FL.BCDN is set to 1 to show that the card image most recently set up is to be punched in BCD mode. Then control goes to PCDE, as it did after setting up an 80-column binary or normal binary card image.

81. This would be the best place to describe subroutine WFR, as all the calls that are made on it have just been mentioned, but what it does when it finds the buffer empty but the status not end-of-record/file has not yet been revealed.

Subroutine WFR

82. This subroutine moves the next k words (where cell WC contains k) in the buffer, to the first of which OUT points, to consecutive PP locations beginning with the one whose address is in the A register on entry. This address is immediately saved at WFR2. Then the return address is saved in D.Z3. Then D.Z5 is zeroed; this is a counter to be increased by 1 each time WFR moves a CM word, until it equals WC, when the normal exit from WFR is taken. With the PP destination address in the A register, subroutine YFR is called to read the next word from the buffer and step OUT. If the buffer is not found by YFR to be exhausted already, the return from YFR is with non-zero in the A register. Then 5 is added to the PP destination address in WFR2 (5 PP words per CM word) and 1 to the CM word counter in D.Z5. If the latter now equals WC, WFR has successfully completed the request to it and exits normally, skipping over the one extra word in its calling sequence. Otherwise return to WFR2 to move the next CM word from the buffer.

83. If the return from YFR is with A = 0, control goes to WFRW; the buffer was already exhausted; now if the status in the FET show end of record/file the special address is gotten out of the calling sequence and a return is made to it, at WFRY. If the FET status is not e-o-r/f, more information is expected to come into the buffer soon, so the attempt to set up and punch this card image should be abandoned, leaving it till the next cycle in the main loop for this file. So an immediate return is made to LPD99. As WFR is called only in the section of 1IR beginning at PCDB, for punching output cards, and since subroutine SAVOUT is not called unless (1) to discard a zero word following a card image containg 40 bytes of information, or (2) after completing a card image, it is clear that the OUT pointer in the FET still points to the beginning of this still-incomplete card

image in the buffer. So the precipitate return to LPD99 will do no harm, even though a false start has been made on a card.

Entry Information

The A register contains the starting address in PP memory of the field to which CM words are to be read from the buffer. WC contains the number of CM words to be read. RJM FIRST; ADD OUT gives the address of the first of the words in the buffer to be moved. The calling sequence is

```
RJM   WFR
VFD   12/a
```

where a is the address to which WFR is to branch if the request cannot be completely fulfilled because end of record/file is met.

Exit Information

OUT has been updated, but not the FET OUT pointer. D.Z5 contains the number of CM words that have been taken from the buffer.

Subroutines Called

YFR

Registers Destroyed

D.Z3

Main Loop of 1IR Continued

84. Control comes to PCDE when a card image of information has been set up, and subroutine SAVOUT is called to update the FET OUT pointer past the information just taken from the buffer.

85. This brings us to PCDK, to which we also came after setting up the image of the first card of a file, or the image of an end-of-file card. These two kinds of card, as well as an information card, are not to be offset, so here FL.PROS is set to 0, and then control goes to PCDKA.

86. We also came to PCDKA, having set FL.PROS to 1, after constructing, in overlay B.EOPCH, the image of one of the two kinds of card that are to be offset, viz. an end-of-file card. At PCDKA, which is thus reached once for each card image whether offset or not, immediately after setting it up in IMAGE through IMAGE+79, subroutine CLERMES is called to set LSTAT to GOOD and clear out any B-display message that may be in the FET. Next,

with 777776B in the A register, subroutine MOVIM is called to save the image of the new card in the one of the two save areas, following the buffer, that was not used for the preceding card. Then 1 is added to the total of cards punched (for accounting purposes) in bits 0-23 of the 7th word of the FET (CT and CT+1). Then FL.IMAGE is set to 1, to show there is an image ready that has never yet been punched, so that if control goes back to LPD99 before punching it, then the next time through the main loop for this file, it will go straight to PCDL without trying to prepare another image.

87. At PCDL, subroutine RES is called to reserve the channel for the punch, select the 6681, and connect the punch. If the exit from RES is with non-zero in the A register, the punch rejected its connect code; the channel and 6681 have been released, and control goes straight to LPD99. Otherwise, subroutine READY is called, which waits until the punch unit is ready and not busy, but waits in such a way as to call LPC if necessary and not hang up other files. At PCDM, the punch is connected and ready, but LPC has probably been called and the image in IMAGE ff. has probably been destroyed, so MOVIM is called with 000000B in the A register, to copy into IMAGE through IMAGE+79 the image that was last stored when MOVIM was called with 777776B in the A register. Finally subroutine PCHNEW is called to punch this image. If the exit is with non-zero in the A register, the punch rejected a function code, and it, the 6681, and the channel have been released; so control goes straight to LPD99. Otherwise, it continues. It may now be necessary to issue an offset function for the card preceding the one just punched, so subroutine READY is called to wait, in an economical manner,until the punch is once more ready and not busy. Then if FL.OS is 0, go straight to PCDN. Of it is 1, it is reset to 0 and then the offset function code, 3 is sent to subroutine FCN to be issued. If this code is accepted, the return from FCN is with 0 in the A register, and control arrives at PCDP to call subroutine REL, to release the channel, 6681, and punch. Otherwise, they have already been released and this call is skipped. It is not attempted again, because after a certain delay it would be ineffective anyway. Now FL.FLIP is inverted, to switch to the other image save area after the buffer, and FL.IMAGE is zeroed, to show there is no longer an unpunched card image waiting. Next, if FL.PROS is 1 (it would have been set so when the image of the current card was set up, if it was an end-of-record card or the blank following an end-of-file card) it is zeroed and FL.OS set to 1 so that on the next trip through the main loop for this file the card just punched will be offset. Then FL.BCDO is reset to 0 or to 1 if FL.BCDN is 1; and FL.BCDN is reset to 0. FL.BCDN causes subroutine PCHNEW to punch in BCD mode, while FL.BCDO does the same for PCHOLD. Finally, go back to LPD99.

88. Observe that because FL.OS was 1, the punch cycle has been completed and control has gone back to LPD99 without checking the previous card for compare error. It must have been either the blank following a 6-7-8-9 card, in which case the check hardly matters, or a 7-8-9 card, in which case it might matter. The excuse for skipping the check on an offset card is that if it were bad, remedial action would result in punching a cluster of offset cards preceding the final offset 7-8-9 card, which would be

confusing. A file coming out of the punch has to have its offset cards checked anyway, to get rid of those that do result from compare errors, and the 7-8-9 cards can be checked for plausibility at the same time.

89. Now if FL.OS was 0, below PCDM just after PCHNEW and READY were called, control comes to PCDN. Here bit 10 of the status byte left by subroutine READY in cell STAT is checked. If this is 0, we can go back to PCDP, just as we did in the case of an offset rather than a check. But if it is 1, the card preceding the one just punched has given a compare error; it has to be offset; then re-punched, hopefully well; then the card just before the offset, which is the significant card here but is now waste paper; then the wasted card is re-punched and goes back to PCDM, as if it had just been punched for the first time, and we repeat the cycle until there is a good check on the card preceding the significant one (the one whose image was most recently constructed).

90. So on detecting the bad bit in STAT, overlay subroutine B.MSG is called to put the message CPnn COMPARE ERROR on the system dayfile (a B-display message is unnecessary here because no operator action is called for; in contrast to the corresponding situation on a card reader, the punch can just grind out cards over and over until it gets them right). Then function 3, offset, is passed to subroutine FCN; if the return is with 0 in the A register all is well and control goes to PCDNA. If the return is non-zero, the function was not accepted and the channel, 6681, and punch have been released. It is no use trying to repeat the function, as after a delay it will be ineffective anyway. But before passing on to PCDNA, where punching will be done again, the channel must be reserved, the 6681 selected, and the punch connected. Before doing this, at PCDNB, LPC is called to let the other files have a chance while this punch is balky. Then call RES; if the return is with 0 in the A register all is well and processing continues; but if not, there is a return to PCDNB to try LPC and RES again.

91. At PCDNA subroutine MOVIM is called with 000001B in the A register. This causes it to copy into IMAGE through IMAGE+79, from one of the save areas after the buffer, the image which is not the one last constructed and saved; i.e. the image of the card preceding the one to be punched, of the card that gave the compare error. Then subroutine PCHOLD is called to punch this previous card. If the return is with non-zero in the A register, the punch rejected a function, and 1IR goes back to PCDNE above to try again. Otherwise, READY is called to wait, while not hanging up other files, until the punch is again ready and not busy. Then FCN is called with 3 (offset function) in the A register, to offset the significant card, even though it may not have given a compare error. If the return from FCN is with 0 in the A register, all is well and 1IR goes back to PCDM, as though about to punch the card in question for the first time. If the return is with non-zero, it is no use repeating the function, which after a delay would be ineffective, but everything has been deselected and released, so 1IR goes to PCDL to call RES and READY before getting to PCDM.

92. Neglecting the possibility of FL.OS being 1, which by-passes checking, and neglecting the possibility that the punch might reject functions, one can say that the series of steps for punching card number n is:

    1. punch card n, and wait for ready and not busy.
    2. if the status is not compare error, we are finished.
    3. if the status is compare error, give a dayfile message.
    4. offset card n-1, which is the one in error.
    5. recover the image of card n-1 and re-punch it.
    6. wait for ready and not busy, then offset card n.
    7. return to step 1, where the image of card n is recovered from central memory before being punched.

93. We have now described everything that concerns punching in particular, except some subroutines that are left to be grouped with subroutines in general at the end of this description.

94. To print a line, we take words from the buffer and store them in the area just after the buffer itself, beginning at LIMIT+1 (i.e. BUFLG-31 words after the beginning of the buffer). This provides a space of 37B words to store the line. The word at LIMIT itself, i.e. the first word after the buffer proper, is used to record our progress in constructing the print line. It has the form:

```
VFD   12/a,12/b,12/c,12/d,12/e
```

where, in the first phase:
a is the number of bytes that have been taken from the buffer and stored in the print line area so far; naturally always a multiple of 5.
b is initially 0, but as soon as FL.LINE is set to 1, showing that line construction has begun, it is set to a program address indicating how far construction has got. On future occasions when we get to LPRQ, if FL.LINE = 1 we branch immediately to this address, b, in KEY+1. Usually it is set by calling subroutine SETBR, which sets KEY+1 to contain the address next after the RJM SETBR call, and then returns to that address.
c is set to contain the first byte of the line to be printed, as soon as it has been read from the buffer.
d is set to contain the first byte of the next line, as soon as it has been read from the buffer, because this information is also needed to decide what skip functions to give before and after printing.
e is not used during the first phase.

When the whole line has been gathered, we begin the second phase and signal this by adding 4000B to a. Then we look at the format characters of this line and the next one, and replace c, d, and e by a series of 6-bit printer function codes, using 70B, 71B, or 72B to indicate actual printing. As each of these codes is obeyed, the 36-bit field is shifted 6 bits leftward, and when the whole field is zero, processing of the line is complete.

In cases where the print line image area has been set up not by copying from the buffer, but by a part of the program that has to add a message to the print file, byte a is set to 6000B + the length of the message. The 6000B means the same as 4000B, concerning how the remainder of this control word is to be interpreted, but also means that the line is in display code. If the printer is a 512 working in extended array mode, the overlay B.TRANS will have to be called, before printing, to translate the line from display code to extended array code, and then subtract 2000B from a.

95. After storing the number of CM words in the buffer in WC, we branch from below LPD1E to LPR if TYPE shows the file is being printed. At LPR, we check the status of the FET. If it is 3 (INITA) or less, it must be the initial value 3, and we have not even set up the first initial page image; so we call overlay B.LINIT, to set up this image, at LPRN. But first, for a 512 printer, we call subroutine IMFIL to initialize its print image buffer.

96. If the FET status is more than 3 but not more than 7 (INITB), it must be 7, and the first initial page image has already been set up. If WC contains 0, we have printed all of that image, and return to LPRN to call B.LINIT and set up the second initial page image. If WC does not contain 0, we go to LPRQ to continue printing whatever is in the buffer.

(One might ask why we call B.LINIT twice for two page images instead of having them both put into the buffer with one call. The answer is that one page image occupies more than half of 340B CM words, though less than 340B. So for the minimum value of BUFLG, 400B, giving an effective buffer size of 340B words, one but not two page images can fit at once into the buffer.)

If the FET status is more than 7, we do not have to worry about setting up initial page images any longer, and branch to LPRO. See section 104 below.

97. In overlay B.LINIT, overlay subroutine B.FCIA is called first; it is really just a table of visible character images. Calling B.FCIA gets the table into the high end of overlay storage, and when we enter the overlay as a subroutine we exit immediately without doing anything. Next we zero IN, in which we shall count the number of CM words put into the buffer, and set OUT to contain address IMAGE. This use of IN will be compatible with its normal use, and OUT will be set to 0 by overlay subroutine B.SETP, called at the conclusion of B.LINIT.

98. The page image is to begin with a page skip and 3 blank lines. A CM word in the buffer with 21B (display code Q) for its leftmost character and zero for the rest will produce a selection of 6-line spacing (in case the printer is a 512), a reset of auto page ejection, a page skip, and a blank line. A CM word with 46B (display code minus) for its leftmost character and zero for the rest will produce an extra double space and a blank line, i.e. the equivalent of three blank lines. So one of the former and ten of the latter will do the first part of our business.

Subroutine LIND is called with 11 in the A register; it begins by saving this in D.Z3, as the number of CM words to be constructed; then multiplying by 5 and storing the result in D.Z2 and D.Z1, as the number of bytes they contain. Then it zeros that many bytes, beginning at IMAGE; then goes back to reset the first byte of each group of five to 4600B. Then it resets the first byte of the whole group, at IMAGE to 2100B, for the initial page skip; then alters the instruction at LINDA so that when we call LIND for the second time during the construction fo this page image, the change from 4600B to 2100B will not in fact happen. Then we call subroutine LINC, with the number of CM words in the A register, to add them to the page image in the buffer.

99. Now the next 12 lines of the page image are to provide a visible copy of the 7 characters of the job name, made out of dollar signs and blanks. These 12 are actually 6 pairs of lines, as each character image is an array 6 high and 10 wide, with each row doubled on the page to make the image tall rather than squat. Each character is 10 wide, and a gap of two columns after each character makes 12 columns, or 6 bytes per character, or 42 bytes for the whole name. An extra pair of blanks before the first character image, and two extra pairs after the last, makes 45 bytes. Following all this on each of the 12 lines, should be the job name in ordinary print; so it is now copied from the first word of the FET to IMAGE+45 through IMAGE+48, with a zero byte in IMAGE+49 to terminate the line; the rest of each of the 12 lines will be constructed in IMAGE through IMAGE+44. Now D.Z1 is zeroed, in which to count lines in the picture; however, it will count up to 6 rather than 12, as after constructing each line we put it into the buffer twice.

100. This brings control to LINAD, where D.T0 is set to point to the beginning of a line, blanks are put in the first byte of the line, and we step D.T0. Now in any of the entries in table FCIA, there are 6 rows of 10 columns represented in such a way that 4040B,4040B,4040B,4040B would be dollar signs in all 10 positions of the top row and blanks elsewhere; 0101B,0101B,0101B,0101B,0101B would be dollar signs in all 10 positions of the bottom row and blanks elsewhere; 7700B,0000B,0000B,0000B,0000B would be dollar signs in all six positions of the leftmost column and blanks elsewhere; and 0000B,0000B,0000B,0000B,0077B would be dollar signs in all six positions of the rightmost column and blanks elsewhere. So to extract from a byte in the table, using a mask 0101B, the relevant bits for two adjacent columns in the current row, first the bytes must be shifted 5 right for the first row, 4 right for the second... and 0 right for the sixth row. So the correct shift is set up at LINAE. Then D.Z3 is zeroed, in which characters taken from the job name in CURFET ff. will be counted.

101. At LINAB begins the loop for each charcter, within the loop for each row. To the current character corresponds a number between 0 and 6, in D.Z3. Characters 0 and 1 are in CURFET, 2 and 3 in CURFET+1, and so on. The character is extracted, its display code value multiplied by 5 and the result left in D.Z4. This is the offset from FCIA for finding the beginning of the table entry for the character. Now D.Z5 is zeroed in which a count

is kept of the number of bytes of that table entry used so far. Then, at LINAC, fetch the next byte from the table, shift it appropriately right, and extract the bits pertaining to the current row using the mask 0101B. Then shift the result left once and subtract it from 5555B; this gives a display code blank corresponding to a 0 bit in the table, and a display code dollar sign (53B) corresponding to a 1 bit in the table. Store the two characters so obtained at the next available position in the line image, to which D.TO points, and step D.TO. Also step D.Z4, to address the next byte of the FCIA table entry for this character, and D.Z5, counting bytes already used. If D.Z5 has reached 5 the entry has been used up as far as this row is concerned; otherwise return to LINAC. After putting 5 bytes into the print line image for this character of the job name in thisrow, put in two more blanks to provide a two-column separation between characters in the final picture, and step D.TO accordingly. Next step the number of the character within the job name, in D.Z3. If it has reached 7, all the characters have been done for this row; otherwise return to LINAB to begin the next character. If all the characters have been done, IMAGE through IMAGE+42 have been filled; now blanks are put in IMAGE+43 and IMAGE+44. IMAGE+45 through IMAGE+48 already contain the job name in display code, and IMAGE+49 is a zero byte to end the line. So now subroutine LINC is called twice, with 10 in the A register each time, to put the 10-CM-word line image beginning at IMAGE in the buffer twice.

102. Next 1 is added to the count of lines completed in D.Z1, and if it has not reached 6, return to LINAD for the next line. At 6, the picture is complete. 18 blank lines must now be added to the page image, which can be done by adding six CM words, each beginning with 46B and continuing with 54 zero bits, to the buffer; such a CM word produces two extra vertical spaces plus a blank line. So subroutine LIND is called with 6 in the A register, to do this. Note that LIND has already adjusted itself so that it does not put a page skip instead of a two-line skip in the first word.

103. Finally, it is necessary to add 5 lines to the page image (the 61st through 65th) each consisting of two blanks followed by 66 dollar signs. Such a line is set up, with a zero byte to terminate it, in IMAGE through IMAGE+44; then subroutine LINC is called four times, with 9 in the A register each time, to add this line image to the page image four times. The page image is complete; everything has been put into it by subroutine LINC, which has kept a running total of CM words in IN . 4 is added to the FET status, and subroutine WTFET is called to make the change in the FET itself as well as CURFET4. INITA is being changed to INITB (3 to 7) after setting up the first page image, or INITB (=7) to 13B after the second. The FET pointers must still be set to reflect the page image at the beginning of the buffer; the length of the image is in IN; overlay subroutine B.SETP (it overlays B.FCIA, which is no longer required) is called to set the FET so that OUT=FIRST nad IN=FIRST + the length in IN, and to set the PP pointer OUT to zero. Finally, we return to LPD1E, where processing of the first line in the page image will begin.

104. We come from LPR to LPRO if both initial page images have already been set up. Now if the buffer contains fewer than 63 words, and the FET status was not end-of-record or end-of-file when we were at LPD1E, subroutine CIO has already been called to refill the buffer. If the status was end-of-record/file, we could not have called CIO there because for a punch file it is necessary to punch an end-of-record/file card before processing the next record. Here, however, we have a print file and can go ahead with CIO.

105. Overlay B.LPEOR is called here if the print file buffer contains fewer then 63 words, and the status is end-of-record/file but not end-of-information. The only complication is that perhaps the current record ends in a CM word whose last byte is not zero, so that the beginning of the next record would appear to continue the last line of the current record. This can scarcely have been intended, so if the buffer is not empty (if it is empty, the above mistake could not occur) and if the last word in the buffer does not end with a zero byte, we call subroutine XFR to add a zero word to the buffer. Whether or not XFR is called, we then call subroutine CIO to refill the buffer, and return to LPD1E to take another pass at this file. The count in WC will not have changed by more than 1, but CIO will have altered the FET status so that it is no longer end-of-record.

106. If B.LPEOR was not called, we come to LPRQ. Now if FL.LINE = 0, we have not begun processing this line and go to LPRA. If FL.LINE = 1, we call subroutine LLRD to copy the print line area from central memory to KEY ff., and branch to the address contained in the second byte of the control word, at KEY+1.

At LPRA, we begin work on a new line by zeroing the control word model at KEY through KEY+4, setting FL.LINE = 1, and calling subroutine SETBR to set the address in KEY+1 to LPRC. Now from LPRC, if the buffer is not empty, we go to LPD14. If it is empty and the FET status is end-of-information, we go to LTER to terminate the file. See section 126 below. If the buffer is empty but the status is not end-of-information, we go to LPRD.

At LPD14 we subtract 1 from WC, for the word we are about to read from the buffer, for the sake of the tests at LPD13D and LPRK below. Then call subroutine YFR, having put the address D.T0 in D.Z2 to tell it to read a word into D.T0 through D.T4. If the return from YFR is with zero in the A register, the buffer was already empty, so we go to LPRD as we did above in the case of empty buffer. Otherwise, to LPD11A.

107. At LPD11A, if KEY does not contain zero, we have already dealt with the first byte of the line, and go to LPD11B. If it does contain zero, and FL.LONG = 1, this is not truly the first word of a line, but merely follows a break we had to make in the middle of an over-long line; so we use 5555B (two blanks) as the first byte of the line for control purposes. Otherwise, we copy the first byte from D.T0 to KEY+2. If it is "PM", we call subroutine SETBR to put address LLDIS in KEY+1, and then go to LLDIS to call overlay B.DIS; this line is to be displayed rather than printed. See section 118 below. If the first byte is not "PM", we go to LPD11C, where we put 0 in the A register and go to LPD11B. This is analogous to the branch from LPD11A, with the contents of KEY in the A register, if those contents were not 0.

At LPD11B we add BUF-1 to the contents of the A register to show, in D.Z1, where to begin storing this word in the line image area. The addend is BUF-1 rather than BUF, because as the CM words are copied into the image, they are shifted one character left in order not to print the first character of the line. The first character will go into the right half of BUF-1, which is KEY+4, where it will be ignored. (It is already in KEY+2 for use as a format character.) The second and third characters go into BUF, and so on. Now we store the ten characters in D.T0 through D.T4 in the right half of the byte D.Z1 initially points to (BUF-1 for the first CM word), the next four bytes, and the left half of the next byte, zeroing its right half (BUF+4 for the first CM word.) When D.Z2 reaches D.T5, 5 is added to KEY, indicating that 5 more bytes have been moved into the line image beginning at BUF. Then if D.T4 contains 0, this was the last CM word of the line, and we go to LPD13. If not, and KEY does not contain 155, we return to LPD14 to get the next CM word. But if KEY contains 155, we call subroutine SETBR to set the address in KEY+1 to LPD13D, and then go to LPD13D, where we see if WC contains 0. If not, we are definitely faced with an over-length line, so we set FL.LONG=1 and then go to LPRH to begin the second phase of processing the line image. If, at LPD13D, WC contains 0, there is a chance that we have just picked up the last CM word of the record. So we go to LPRE and check for end-of-record status; if yes, go to LPRH without setting FL.LONG=1, as the next line will be at the start of a new record and its format character will be a proper one. If not end-of-record, it is still possible that the FET status may become e.o.r. without any additional words being read, so go to LPRG to copy the print line image back into CM, update the OUT pointer in the FET, and return to LPD99. The next time we pick up this file, we will branch from LPRQ to LPD13D to see if further reading of the file has resolved this point.

108. We come to LPD13 when we have found a proper line terminator. We call subroutine SETBR to set the address in KEY+1 to LPRK, and then, at LPRK, see whether the buffer is empty. If not, we read (but without altering the OUT pointer as subroutine YFR would; i.e. we are peeking ahead here) the next word in the buffer, which is the first word of the next line, and put its first byte in KEY+3 for use in format decisions. But if the buffer is empty, we go to LPRE and check the FET status. If end-of-record, we need not consider the first byte of the next line, so we go to LPRH and continue. If not end-of-record, however, we go to LPRG to store the line image and control word back in CM, update the OUT pointer in the FET, and return to LPD99. The next time we pick up this file, we will branch straight from LPRQ to LPRK to see whether further reading has either found e.o.r. with no further information, or has brought the first word of the next line to the buffer.

109. We come to LPRD if, before finding a line terminator or reaching the limit of 155 bytes, we find empty buffer. Then if KEY contains zero, we have not really begun preparing the line image, and so can just reset FL.LINE to 0 and go to LPR99 to update the FET OUT pointer and return to LPD99.

But if not, we go to LPRE and check for end-of-record status. If yes, we let it mean the end of a line, so we branch to LPRH. If not e.o.r. status, we go to LPRG to write the line image and control word back to central memory, update the FET OUT pointer, and return to LPD99. The next time we pick up this file, we will branch from LPRQ to LPRC, to see if further reading has enabled us to go further with this line.

110. We come to LPRH when we have taken all we want or all we can get from the buffer for this line, and are ready to begin the second phase. Call subroutine SAVOUT, to update the FET OUT pointer, and zero D.T1 through D.T5. We are going to prepare a new control word in D.T0 through D.T4, and the counter initialized in D.T5 will be used in this. The first byte, D.T0, is formed by adding 4000B, the second phase indicator, to the line length in KEY. The second byte is set to address LPRL, so that on all future occasions when we pick up this file for this line, we will branch from LPRQ to LPRL. At the moment, we are going to complete the new control word without interruption and pass then to LPRL.

Now if FL.SU=1, format control on this file has been suppressed by operator type-in, and we are to behave as though every line began with a blank format character, so we go to LPD40. We also go to LPD40 if KEY+2 and KEY+3 show that both this line and the next began with blanks. This is the simplest, and hopefully commonest, case. At LPD40, we set KEY+2 to contain 7000B, so that the new control word is:

```
VFD     12/4000B+a,12/LPRL,6/70B,6/0,6/0,6/0,6/0,6/0
```

where a is the length of the line image. The six-bit fields represent printer functions, from left to right, and 70B means print the line image. So beginning at LPRL we shall just print it. From LPD40 we come to LPRW, call subroutine LLWT to copy the old control word and the line image back to central memory, and then write the new control word from D.T0 through D.T4 on top of the old one. That brings us to LPRL.

But if we do not go to LPD40, we probably have a more complicated format action required, and must call overlay B.FORMAT. This will set up a control word:

```
VFD     12/4000B+a,12/LRPL,6/b,6/c,6/d,6/e,6/f,6/g
```

where a is the length of the line image, and b through g are printer action codes to be executed from left to right, with unused slots occupied by zero. Then return to LPRW as above.

<u>Overlay B.FORMAT</u>

111. Note first of all that any pre-print skips, other than skips to top or bottom of page, will be achieved as post-print skips on the preceding line; or if the preceding line already has a post-print skip, a one-word blank line will be interpolated to carry the pseudo-pre-print skip. This is logically less simple than using the printer pre-print skip functions, but in the end it usually means fewer hardware functions actually sent to the printer.

No hardware functions can be done in an overlay, as they involve waiting. So in this overlay we accumulate a list of hardware functions to be done afterwards. Each function is represented by its actual hardware code; except that printing the line is represented by 70B; printing a one-word blank line (as a way of interpreting a pre-skip after a post-skip) is represented by 71B. 70B followed by 10B is more compactly expressed by 72B; this is necessary whenever the 70B follows a post-print skip selection on a 501 printer, though not on a 512 printer, in order that the 10B function may clear the selection after the skip has taken place. This list of things to be done is kept in bits 0-35 of the control word in the print line storage area; i.e. in KEY+2, KEY+3, and KEY+4; but in this overlay we are initially constructing it in D.T2, D.T3, and D.T4, which have been pre-zeroed, and a count of how many functions have already been inserted is maintained in D.T5, initially zero.

112. Subroutine DO is used to add a function to the list and if necessary to charge for it. On entry, the function is in bits 0-11 of the A register, and the number of lines to be charged for it is in bits 12-17. The former is saved in D.Z2, and the latter is added to the total line count in bits 0-23 of the 7th word of the FET. D.T5 contains the number of functions already in the list; if we express this as 2p+q, where q is 0 or 1, the new function is to be stored in the left (for q=0) or right (for q=1) half of D.T2+p. After storing it there, we add 1 to D.T5.

The 36-bit space restricts us to not more than 6 functions for a single line, but this is more than enough at present because of the use of the combination code 72B for 70B followed by 10B.

So to begin doing this overlay, at FORMATX, we get the format character of the current line from KEY+2. If it is Q, we should reset auto page skip, do a page skip unless the next line already calls for one, and not print the current line. So we set FL.AUTO=0, and pass a reset function through DO, 10B for a 501 printer or 30B for a 512. In case of a 512, we also have to consider that the reset will have selected 6-line spacing as well as resetting the auto page skip. If the file was supposed, in fact, to be in 8-line spacing, FL.8LL will =1, and in that case we pass a 10B function through DO; for the 512 this means select 8-line spacing. Then go to SPCV to deal with the first character of the next line.

If the first character of this line is not Q, we go to SPCA and check for R. R means we should not print the line, but do auto page skipping till further notice, and also do a page skip before the next line. So we set FL.AUTO=1. This will remind us, whenever we do a reset function for a reason other than clearing auto page skip, that we must renew the auto page skip function. Then with 5, the auto page skip function code, in the A register, we go to SPCVA to send that code to the printer, and then, at SPCV, deal with the first character of the next line.

At SPCV, if the first character of the next line is found to be 1, this will produce a page skip when that line is processed, so we go back to LPRW. Otherwise, we first provide a page skip by putting 4, the function code, in the A register and going to SPCN, where we shall pass the code through DO and then deal with a possible pre-print skip code in the next line.

If the first character of the current line is not Q or R, we go to SPCB, and if S or T, go to SPCP. These characters are for not printing the line, but selecting 6-line and 8-line spacing respectively; but if FL.512 = 0 they are irrelevant and we go back to LPRW. Otherwise, we set FL.8LL = 1 for 8-line or 0 for 6-line spacing; pick up the function code, 10B or 11B, and branch to SPCN to put it through DO and then deal with a possible pre-print skip code in the next line.

If the first character of the current line is not Q, R, S, T, A, or B, we branch to SPCC. For A or B, meaning post-print skip to top or bottom of page, we successively pass through DO a 6, for suppress normal spacing after print, and 70B, for printing the line; then we put 4 or 3, for skip to top or bottom, in the A register, along with a change of PC.SKIP lines, and go to SPCN, where we pass the code through DO and then deal with a possible pre-print skip code in the next line. For post-print skip to top or bottom of page, there are specific functions in the 512 printer but not in the 501 printer, so for both printers we handle it indirectly as above.

113. At SPCC, we see if the first character of the current line is C through L, calling for a post-print skip. If not, go on to SPCE; if so, with 0 through 11B in the A register, representing C through L, call subroutine POST to add the proper post-print skip function, if any, to the list. On return from POST, the A register contains 2 if a post-print skip function for the 501 printer was added to the list, and otherwise zero or 777777B. So we add 70B to this and pass the result through DO; i.e. 70B for a simple printing of the line, or 72B for a printing followed by a 10B function to clear the post-print selection on the 501. This brings us to SPCN.

At SPCN we put a function, which has in every case been mentioned above, through DO and then consider the first character of the next line via subroutine EV. In general, where the format character of a line (the next one) calls for a pre-print skip, we would like to accomplish it by a post-print skip on the preceding line (the current one), and this is how we shall do it at SPCM below. But for all the cases considered so far, either the current line is not to be printed anyway, because it is merely a carrier for format character Q, R, S, or T, or it called for its own post-print skip with a format character A through L. Now if the return from subroutine EV is with the A register negative, the first character of the next line is not 3 through 9 or X through Z, i.e. does not call for a pre-print skip, and we have nothing further to worry about; so we exit from the overlay to LPRW. If the return is positive, however, EV has translated the format character of the next line into a number between 0 through 11B, and with this we call subroutine POST to set up the corresponding post-print skip code, if any, and send it through DO. If the return from POST is with 0 in the A register, no post-print skip code was sent through DO (there are fewer channels on the 501 format tape than on the 512, so some format characters turn out to be simply inapplicable for the 501) and we can exit to LPRW. But otherwise, we must supply a blank line for the post-print skip to be applicable to, as it certainly must not apply to the next line of the text, which is calling for a pre-print skip. So we send 71B, calling for a one-word blank line to be printed, through DO and then go to LPRW.

We come to SPCE if the first character of the current line is not Q, R, S, T, A, B, or C through L. If it is 3 through 9 or X through Z (pre-print skip) we are not interested in it, because the proper equivalent action was set up when the preceding line was processed. If it is 1, we take 4 (skip to top of page function) to SPCJ unless FL.PAGE = 1, in which case go to SPCG. When FL.PAGE = 1, the next line to be printed will be preceded by a page skip anyway, so there is no point in interpreting its first character as another page skip. If the first character of the current line is 2, we take 3 (skip to bottom of page function) to SPCJ. Along with function 4 or 3 goes a charge of PC.SKIP lines. If it is 0 or -, we take 1 or 2 (single or double extra space function) to SPCJ.

At SPCJ, we put the function through DO and arrive at SCPG. Here we check the first character of the next line to see if it is 1, 2, or +. If any of these, pass a 6 code (suppress normal post-print space) through DO and go to SPCGE. Otherwise go to SPCM. The suppression before + is obvious, and ensures that the next line will be overprinted on the current one. (Note that we did not get here if the format character of the current line was A through L, calling for a post-print skip.) The suppression before 1 and 2 is not so obvious, but is logical in terms of the specifications.

At SPCM we call subroutine EV to see if the next line format character calls for a pre-print skip, which we can accomplish as a post-print skip on the current line. If not, the return from EV is negative, and we go to SPCGE. If so, we enter subroutine POST with the appropriate number between 0 and 11B in the A register. If POST passes a 501 (not 512) post-print skip code through DO, the exit is with 2 in the A register; otherwise with 0 or 777777B; we add 70B to this and pass the result through DO; i.e. 70B, meaning print the current line, or 72B, meaning print the current line and then do a 10B function to clear the post-print selection on a 501 printer. After that, return to LPRW.

If we go to SPCGE, because the first character of the next line does not call for a pre-print skip, we send 70B through DO, for simply printing the present line, and then go to LPRW.

114. Subroutine EV fetches the first character of the next line from KEY+3. If the character is 3 through 9, EV exits with 0 through 6 in the A register. If the character is X through Z, EV exits with 7 through 11B in the A register. These exit values are 3 below the binary value of display codes C through L, which correspond to 3 through 9 and X through Z as post-print skip characters to pre-print skip characters, the exit from EV is with the A register negative.

115. Subroutine POST is entered with 0 through 11B in the A register, corresponding to C through L as the format character of the current line, or 3 through 9 or X through Z as the format character of the next line. This number is saved, and then we branch to the top part of the subroutine (POSTZ) for a 512 printer or continue with the bottom part for a 501. For a 501 printer, if the entry number is 6 or greather (corresponding to format characters I through L, 9, and X through Z, there is

nothing to be done, and we merely exit from POST with 0 in the A register, showing that no post-print select clear will be needed after the next print. Otherwise, for a 501, we turn the entry values 0 through 5, representing channels 6 through 1, into function codes 16B through 11B (select post-print skip to channel n-10B); send the resultant code through DO along with a charge of PC.SKIP lines, and exit from POST with 2 in the A register, indicating the need of a clear function after the next print.

At POSTZ, for a 512 printer, we turn the entry value 0 through 11B into a charge of PC.SKIP lines plus function code 36B, 35B, 34B, 33B, 32B, 43B, 37B, 40B, 41B, or 42B, pass the code through DO, and then exit from POST with 777777B in the A register. This number counts a s non-zero to the ZJN test just below SPCY3 (a 71B "function" will be required) but as zero when it is added to 70B just above SPCN and below SPCGE (as a 512 post-print skip does not need to be cleared after printing.) Functions 32B through 43B are 30B+n, calling for a post-print skip to channel n.

116. We come to LPRL when we have completed the second phase of preparing the print line image and control word, and are ready to start using the hardware. The control word looks like this:

```
VFD     12/4000B+a,12/LPRL,6/b,6/c,6/d,6/e,6/f,6/g
```

where a is the number of bytes in the print line image (the first byte may also be 6000B+a, as noted below), and LPRL in the second byte means that if we go to LPD99 before finishing with the line, we will get to LPR, from there to LPRQ, and from there directly to LPRL without trying to repeat the formatting process. b, c, d, e, f and g are printer functions to be carried out in that order, and the first zero found in the sequence indicates the end of the series of functions.

We call subroutine RES to reserve the channel, select the 6681, and connect the printer. If it fails, the return is with non-zero in the A register and we return to LPD99. Otherwise check FL.PAGE. If =1, this is an overriding call for a page skip before printing the line. So we zero FL.PAGE and put function code 4 (skip to top of page) in DEEP2, where each function is stored before execution. At LPRLC, call subroutine READY to wait, with switching to other files as required, till the printer is ready and not busy. Then pick up the function code and send it to subroutine FCN for execution. If the return is with zero in the A register it was accepted and we go to SPC5; otherwise reserve the printer again and return to LPRLC; or if subroutine RES fails, return to LPD99.

117. We come to SPC5, with the printer reserved, if FL.PAGE=0 or when it has been obeyed. Now, if the printer is a 501, and FL.AUTO=1, and DEEP2 shows that the last function was 10B, we must renew the auto page skip select; so we go back to LPRLB with 5, the proper function code, in the A register. Otherwise, arrive at SPC5N. Here we read the control word from the print line area to D.T0 through D.T4. Then the first, or next, printer function to be done is in the left half of the third byte, at D.T2. If this is 0, we have finished the normal processing of the line and go to SPC57. Otherwise, we store this function code

in DEEP2, and then shift the 36-bit field in D.T2 through D.T4 six bits left, squeezing out the function code just saved in DEEP2, bringing the next one into the left half of D.T2, and introducing a zero at the right end. Then write D.T0 through D.T4 back to the first word of the print line area.

Now pick up the function in DEEP2 and see if it is 70B or greater. If so, it calls for printing a line, and we go to SPC53. Otherwise, it is a hardware function code, and we go to LPRLC to wait for ready and not busy status in subroutine READY, issue the function via subroutine FCN, and return to SPC5 for the next function.

At SPC53, we store the code minus 70B in DEEP2. Now before printing, either (a) for a 501 printer call READY to wait (with file switching) till the printer is ready and not busy, or (b) for a 512 printer call RUDDY to wait (with file switching) till the printer is ready and memory not busy. Then, at SPC53B, see if the code was 71B or not. If not, go to SPC53A to print the line image; but if so, we must now print a minimal blank line. This is done by setting the byte count in WC to 1, and making the byte in BUF=0000B, i.e. two blanks, and going to SPC51. At SPC53A, on the other hand, we call subroutine LLRD, which reads the control word into KEY through KEY+4 and the line image into BUF ff.; then set WC to contain the byte count in KEY. If the 2000B bit of KEY is 0, we can go straight to SPC51 to print; but if not, this is a line that was created by the program as a message. If the disposition code is 44B, we are printing this file in extended array mode, and must, before going to SPC51, call overlay B.TRANS to translate the message from display code to extended array code (one byte per character).

Overlay B.TRANS

117A. As we are going to translate from two characters per byte to one per byte, the byte count in WC has to be doubled, but not above 155. So we reduce the count to 77 if it is too large, and then put the same old byte count in D.Z1 and twice it in D.Z2 and WC; and set KEY to 4000B+the new byte count rather than 6000B+ the old one.

Working from right to left, to avoid putting new data on top of unused old data, we translate the 2x display code characters in BUF through BUF+x-1 into 2x extended array characters in BUF through BUF+2x-1. If the binary value of a display code character is y, its corresponding extended array character is in cell TRATAB+y. Table TRATAB is not now coded in the overlay, as extended array printing is not possible yet, but the table will have to be provided eventually. As the messages the program provides are the only thing that has to be translated like this, and as they contain no characters whose display codes are above 57B, only 60B bytes of table have to be provided.

Having translated the line, we call subroutine LLWT to copy the new version of the control word and line image into the print line area. Then go straight to SPC51 to print the line, as the new version of the line is in BUF ff., and the new byte count is in WC.

117B. At SPC51, we are ready to print the line. But first we suppress trailing blanks, to save time, and also to avoid trouble with lines formatted as 140 characters but having four or more blanks at the end. LDM BUF-1,WC picks up the last byte of the line. If this is not 0000B, 0055B, 5500B, or 5555B, we assume it is not two blanks, and go to SPC55; all done. If the byte is one of these, we eliminate it by subtracting 1 from WC and try again. If WC reaches 0, we reset it to 1 and go to SPC55, so as not to make the line vanish. The preceding test will work for extended array printer code if the trailing blanks are represented by 0000B, but not if by 0040B, which is the genuine hardware code for blank on the 512 printer in extended array mode.

At SPC55, we add 1 to the total of lines printed from this file, in bits 0-23 of the 7th word of the FET.

At SPC52, we call subroutine PRT to print the line. Then call subroutine SALARM to set the alarm for this file to now +AL.LP msec., as the time before which the printer cannot possibly be ready and not busy, and it is pointless to start work on the next line. Next, if the printer is a 512, go to SPC59. But if it is a 501, we may have to issue a 10B function to clear a post-print skip selection on the line just printed. If DEEP2 contains 0, the "function" we just did was 70B, and we don't have to, so we return to SPC5 to get the next function for this line. But if DEEP2 contains 1 or 2, the function was 71B or 72B. 71B was for a blank line created specially to carry a post-print skip, representing a pre-print skip on the following line, so a 10B function is needed. 72B was for printing the current line, but its use instead of 70B indicates that it was preceded by a post-print skip function. So we put 10B, the clear function code, in the A register, and return to LPRLB to execute it before going to SPC5.

117C. At SPC59, to which we come after printing a line on a 512 printer and setting the file alarm, we call RUDDY to wait for the printer to be ready and memory not busy, and then check the printer status in STAT. The status is left there by subroutine STS, which is called by RUDDY; no switch to another file can have taken place between the time RUDDY found ready and memory not busy status, and the return from RUDDY to here. Now we check the two status bits that indicate printer function error; if both zero, go on to SPC60. If either fault is present, call subroutine REL and then subroutine IMFIL to refill the printer image buffer, which may have deteriorated somehow and caused the error. REL has to be called before IMFIL because IMFIL includes a call to subroutine RES. Now in all the code between SPC53 and RUDDY-1, we assume that the printer is always reserved; in fact it is not, but the

times when it is not are hidden in the calls to READY. But since IMFIL expects to find the channel, 6681, and printer unreserved, unselected, and unconnected, we must call REL before calling it.

After calling IMFIL, we call subroutine MAYMES to set LSTAT to WROGER, to put the message "PRINT ERROR" on the system dayfile, and to put the message "----PRINT ERROR----" on the B- and J-displays. Then abandon this file for the moment by branching to LPD99. The fact that LSTAT contains status WROGER will, at LPD1A, prevent any further work on the file until the operator types in /OK for this printer. Before doing so, he may use a /BS type-in to backspace this file one or more pages, hoping for an error-free reprint of the page containing the line that gave the print error.

If, for a 512 printer, neither print error status was found, we come to SPC60 to make a final check, for unprintable characters in the line just printed. See the description of subroutine PRT for the meaning of "unprintable" in this sense. If no such characters were found by PRT in this line, FL.CMESA=FL.CMESB=0, and we go to SPC5 to see what the control word calls for next. Otherwise, we call overlay B.CMES to print an unprintable character message immediately below the offending line.

Overlay B.CMES

117d. Either FL.CMESA for FL.CMESB is = 1, but not both. If the former, we have now to print, right under the line that contained one or more unprintable characters, a line consisting of a blank under each printable character and an X under each unprintable character. FL.CMESA was set = 1 when the unprintable character(s) were discovered. If FL.CMESB=1, we have already printed the line just described, and must now print the message "***** UN-PRINTABLE CHARACTERS IN PRECEDING LINE ****. This being the simpler operation, it will be described first.

If FL.CMESB=1, we know FL.CMESA was reset to 0 on the preceding call of the overlay, and now reset FL.CMESB. Then we copy, into the line image area, the control word and message beginning at CMES. The first byte of the control word is 6043B, indicating that the print line is 43B bytes long, that we are to treat it as having already gone through the first phase of preparation, and that it is in display code, so that in case this is a file being printed by a 512 in extended array mode, overlay B.TRANS will have to be called at SPC53A to put the message into extended array code. The second byte of the control word contains address LPRL, which is normal for a line image in the second phase. The third byte contains 7000B, indicating that the first printer action is to print the line (70B), and then no further action is needed (00B). Having copied the control word and message into the line image area, we branch to SPC5, where we shall read it back up from the line image area and get to work on it. Note that the printer is already ready and not busy.

117e. If FL.CMESA=1, we go to CMESA, reset it to 0, and set FL.CMESB=1 for the next trip through this overlay. Then call subroutine LLRD to read the control word and line image of the offending line from the line image area, extract the length of the line from KEY, and store this in D.Z1 and D.Z3. Then set the second and third bytes of the control word, at KEY+1 and KEY+2, to LPRL and 7000B for the same reasons as in the preceding paragraph. KEY+3 and KEY+4 are zeroed, but this is probably not necessary. Now we must be printing this file with a 48-character set or a 288-character set, as a 64-character set could not consider any display code character unprintable. We assume that the disposition code for the 48-character set would be 43B, and thus branch to CMESE for 48 characters or CMESF for 288 characters.

Beginning at CMESF, we scan the line image byte by byte, and replace any byte greater than 289 by 0030B, and any other byte by 0000B. Then at CMESH, we work through the line image from left to right, discard the left half of each byte, and combine the right halves of pair of adjoining bytes into a single byte, thus halving the length of the line image. By this extra step, we get a line image of display code X's and blanks that will get expanded later back into extended array code by overlay B.TRANS. We seem to be doing a useless step here, but in fact we do not know, in general, what the extended array code for X will be; so it is easier to set up the line in display code and leave the decision to B.TRANS. We determine when we have finished this conversion by subtracting 2 from D.Z3 for each pair of bytes combined into a single byte. If D.Z3 reaches zero, there was an even number of bytes, and no problem. If it turns negative without passing through zero, there was an odd number, and the last pair of bytes treated consisted of the last genuine byte and a spurious one. So at CMESJ, we zero the half-byte that resulted from the spurious byte. In either case, on reaching CMESK, we have the number of new bytes in D.Z5, and add this to 6000B to give the proper first byte of the control word, in KEY. Then, at CMESG, call subroutine LLWT to copy the control word and line image back into line image storage, and branch to SPC5 to work on it.

At CMESE, we can work entirely in display code, and scan the line image character by character, replacing any byte greater than 61B by 30B, for X, and any other byte by 00B, for blank. When this is complete, we can leave the first byte of the control word, in KEY, unchanged, and we merely branch to CMESG to call LLWT and then go to SPC5.

Note that if, when the line containing unprintable characters was printed, there were still one or more functions in the control word left to be done (some sort of programmed skip, or possibly a post-print skip selection clear), these functions will not get done because the control word is partly destroyed by overlay F.CMES.

117f. We reach SPC57, from just below SPC5, when we have completely carried out all the printer functions for the line, or if we printed the line, found it to contain unprintable characters, and printed the message about that. Now, unless FL.PRSU = 1, we have completed processing of this line, and can reset FL.LINE and exit to LPD99.

If FL.PRSU = 1, however, it was set because of a type-in to indicate that at the next opportunity we should do a special page skip, print **** FORMAT CONTROL SUPPRESSED **** and then set FL.SU to cause format control characters to be ignored for the rest of the file. So we branch to SPC57A and there call overlay B.SUP.

117g. Overlay B.SUP

We begin by resetting FL.PRSU and setting FL.SU = 1, so that format characters will be ignored for the rest of the file. Then copy the control word and line image beginning at SUPMES into line image storage. The first byte of the control word is 6036B, indicating that the line image is 36B bytes long, that it is to be treated as having already gone through the first phase of formatting, and that it is in display code, so that in case of extended array printing overlay B.TRANS will have to be called at SPC53A to put it into extended array code. The second byte contains the address LPRL, which is normal for a control word during the second phase. The third and fourth bytes contain 04700000B, which taken from the left to right means a page skip, followed by the printing of the line, followed by no further printer action. Now branch to SPC5, where we begin treating all this as if it were an ordinary print line.

118. We call overlay B.DIS at LLDIS on finding that the print line begins with "PM". Before calling the overlay, we call subroutine SETBR to set KEY+1 to contain address LLDIS; so that if, in this trip through the overlay, we exhaust the buffer before reaching the end of the line, then the next time this file is picked up control will go from LPRQ straight to LLDIS.

On entering the overlay KEY is tested and branch made to DISA for a line that has already been begun. If on the other hand the line is just now being begun, KEY is set to contain 5 and the first five bytes of the line, just read from the CM buffer to D.T0 through D.T4, are copied into BUF+1 through BUF+5. Instead of dropping the first (format) character, as for printing, we should drop the first two (PM). The display line will begin in BUF, but other information is going to be put in BUF and BUF+1, so storing the line beginning in the left character of BUF+1 will cause its first two characters to be effectively dropped.

119. When control comes to DISA,KEY contains a number n, and the line so far has been stored in BUF+1 through BUF+n. So BUF+n+1, the address of the cell in which the next CM word from the buffer is to begin, is put in the A register and D.Z2, and subroutine YFR is called to copy the next word from the buffer. Normally, YFR exits with non-zero in the A register, and there is a branch to DIS11, where a check is made for the last byte of the CM word to see if it terminates the line. If so, we go to DIS13; otherwise, 5 is added to KEY and if it does not now contain 155 a return to DISA is performed to continue the line. If it does contain 155 the line must be cut off, and FL.LONG set to 1, for the same reasons described in connection with LPD13D above; then control goes to DIS13 as if a proper terminator had been found.

120. If the return from subroutine YFR, below DISA, is with zero in the A register, D.Z2 is reduced by 5 so that it will point to the first of the last 5 bytes that were read previously. Then 1IR checks whether CURFET4 shows end-of-record. If so, it goes to DIS13 just as if the last word brought from the buffer had ended in a zero byte. If not, the buffer is exhausted for the moment but there is more of the record to come, so we go to LPRG (in the permanent part of 1IR) just as in the parallel situation below LPD14.

121. We come to DIS13 when the end of the line has been reached, one way or another. First subroutine SAVOUT is called to update the OUT pointer in the FET. Then FL.LINE is set to 0 to show there is no longer anything interesting in the 40B CM words that follow the buffer. When DIS13A is first reached, D.Z2 still contains the address that was in the A register for the last entry to subroutine YFR that read a CM word; i.e., it contains x, where x+4 is the address of the last byte in the display line. So 5 is added to D.Z2, giving the address at which we have to start filling in zero bytes. Then at DIS15 zero bytes are read into the next five cells, and 1IR returns to DIS13A to repeat the cycle. But when the address in D.Z2 reaches or surpasses BUF+26, it goes on to DIS14, as nothing after BUF+24 will be used.

122. Now the message has been taken from the print file for display in BUF+1 through BUF+24. BUF+1 contains only the signal characters PM, so the message really runs from BUF+2 through BUF+24. Any of the line that has been read from the buffer by YFR into cells beyond BUF+24 is simply lost, while if the message is too short to fill the space,zero bytes have been used as padding. There is a danger that one or more of the bytes BUF+2 through BUF+24 may have a numerical value 6000B or greater, so that it would be treated by the scope hardware not as a pair of characters but as a coordinate. So at DIS14 and DIS14A these bytes are scanned and the contents of any byte that are not already less than 6000B are reduced by 6000B.

123. The 9th through 14th words of the FET are the J-display area. Permanently, the 9th word and the first two bytes of the 10th contain the equipment type, the EST ordinal, and the file name. Ordinarily, the third byte of the 10th word contains 0000B; if not, the 23 bytes beginning at that one and ending with the end of the 14th word constitute an addition to the J-display for the file, which will also be displayed on the B-display if one of the four lines is available. So now we copy the 23 bytes, BUF+2 through BUF+24, into the 23 bytes in bits 0-35 of the 10th word, and all of the 11th through 14th words,of the FET.

124. It is still possible that the line in the print file consisted of nothing but the two letters PM, followed by blanks. If so, it will have come to 1IR as one CM word with PM in the left byte and 0000B in each of the other bytes. The line is not worth bothering over further; and in fact both BUF+2 and the middle byte of the 10th word of the FET will contain 0000B, so that no line on the B-display will be produced. So if BUF+2 contains 0000B, control goes straight to LPD99. Otherwise, PC.DIS

(=1000) is added to the count of lines printed, in bits 0-23 of the 7th word of the FET. This indicates, for accounting purposes, a nominal wait of one minute by the printer, while the operator obeys the displayed message. Then LSTAT is set to contain WROGER (=5). This means that the file is hung up, with the message presumably on the B-display, until the operator types in /OKpp., where pp is the EST ordinal of the printer as it appears in the message itself. The hang-up is assured at LPD1A; the clearing, on recognition of the OK type-in, occurs at OK. Then control goes to LPD99.

Terminating a Print or Punch File

125. For a punch file, we recognize at PCDC that the file has been completed when the FET status is 1013B and the buffer is empty, and go to MTER. There overlay B.MTER is called, which 1) if the repeat count is not zero, reduces it by 1, rewinds the file, and returns to LPD99 2) if the repeat count is zero, branches to TER in the permanent part of 1IR, where overlay B.TER is called to terminate the file.

126. For a print file, we recognize at LPRC that the file has been completed when the buffer is empty, FL.LINE is 0, and the FET status is end-of-information. Then we branch to LTER, where, in the first instance, we will branch to MTER because FL.FMES will be 0. At MTER overlay B.MTER is called, which, as for a punch file, decides whether to repeat the file or not. If so, it rewinds the file and branches to LPD99. If not, it puts a message //// END OF LIST //// preceded by the job name, into the line image area, sets FL.FMES to 1, and branches to LPD99, so that this message can be printed normally. After the printing of the message, control will again get from LPRC to LTER, but this time we branch to TER and overlay B.TER is called because of FL.FMES being 1. B.TER will really terminate the file.

Here are the details of the two overlays:

B.MTER

127. We begin by branching to MTERK if the repeat count in bits 42-47 of the 7th word of the FET is 0, as the file has been processed for the last time. Otherwise, subtract 1 from this field, and according to its new value, 0 1 2 3 4 5 6 or higher, alter the character in bits 42-47 of the 10th word of the FET to blank - = * / + (or). This character immediately follows the seventh character of the file name as provided for the J and B displays. - = and * are chosen to represent 1, 2, and 3 repeats remaining because in their scope display forms they have 1, 2, and 3 strokes. Then we call overlay subroutine B.REW to rewind the file. Then, for a punch file, we go back to LPD99; but for a print file we first put a blank word with format character Q into the buffer, and set FL.LINE and FL.8LL to 0. This ensures that before starting to re-read the file we shall do a page skip and reset the auto page eject flag; and in case of a 512 printer, select 6-line spacing.

128. At MTERK, for a punch file, we branch immediately back to TER in the permanent part of LIR, so that overlay B.TER is called to terminate this file. But for a print file, the final message still has to be provided. This is constructed by fetching the job name from CURFET through CURFET+3 and putting it between 20 blanks on the left and the words /// END OF LIST /// on the right. Then this message, preceded by the control word beginning at TERMITE, is written into the line image area. Then we set FL.AUTO to 0 and set FL.LINE, FL.PAGE and FL.FMES to 1, and return to LPD99. The next time we pick up this file, the byte = LPRL in the control word and the fact that FL.LINE = 1 will send control to LPRL, where FL.PAGE will demand a page skip. The first byte of the control word, 6036B, will show that there are 36B bytes in the line to be printed, and that they are in display code and so must be converted if the printer is in extended array mode. After this line is printed, the file will again be in terminal status, but the fact that FL.FMES = 1 will cause B.TER to be called for actual termination.

B.TER

129. This begins by calling overlay subroutine B.DROPBF to release the buffer that was used for the current file. Then subroutine CLR is called to zero IMAGE through IMAGE+79, and this area is copied into central memory to zero all 16 words of the FET. Then the punch unit or printer, whose EST ordinal is in PSTAK,PS, is dropped.

130. Next the disk storage space that the file occupied must be released. For this a stack request is necessary:

```
VFD     12/z,24/0,12/0.RCHN,6/0,6/pu
VFD     36/x,6/14B,18/x
```

where z is the initial RBT pointer from bits 36-48 of the second word of the FNT entry, and pu is the physical unit number taken from the first word of the RBR to which the first RBT word-pair points, and x does not matter. 6/14B in the second CM word means that neither an FET nor an FNT entry is to be used by stack processor, so that the FNT can be destroyed without waiting for fulfillment of the request.

131. So the FNT entry address is taken from DEST3 and subroutine PRERQ called. This partly sets up the request as follows, in MYRQ through MYRQ+9:

```
VFD     12/0,12/f,12/0,12/y,6/0,6/pu
VFD     18/0,18/first,6/0,18/limit
```

On return from PRERQ, the A register contains 0 and we zero 12/f in the request. As PRERQ has left a copy of the FNT entry in MYFNT through MYFNT+14, the initial RBT pointer in MYFNT+6 is examined. If this is zero, the file was empty anyway and we go to TERB without bothering to make the request after all. Otherwise, it is inserted in the first byte of the request; the flags are added in the second CM word (6/0 becomes

6/14B), and subroutine DORQ is called with the function code O.RCHN in the A register. DORQ inserts this in the first CM word of the request in place of 12/y and gets the request issued.

132. Then at TERB the FNT entry is destroyed, taking advantage of the fact that IMAGE through IMAGE+79, have already been zeroed though MYRQ and MYFNT have overlaid the first 25 PP words of this area. Then overlay subroutine B.AMSG is called to issue a dayfile accounting message for the file. Then zero PSTAK,PS to show that the FET is free. Then set FNTALARM 512 seconds ahead of the current reading of the seconds clock, to ensure that on the next trip through subroutine LPC, overlay B.FNT will be called to try to get the FET and the printer or punch back into use; then jump to LPCK in subroutine LPC, instead of entering LPC normally, because nothing should be stored back into what is no longer an active FET.

Card Reading in the Main Loop of 1IR

133. We branch from START7 to CRD if we have begun the main loop, taken care of type-in and buffer-moving possibilities, and then found that a card-reading file is to be processed. If DEST3 does not contain 0, we go to CRDA, but if it does, the address of the FNT entry is 0. This means there is no FNT entry, but there is an FET, a card reader, and a buffer; after reading the 6-7-8-9 card that terminated the preceding file in the same card reader, we found that the reader was not empty, but that it had physically read the next card (the job card of the file it is now to work on) without really looking at it. To save labor, we kept everything but the FNT entry, which had to be converted from the entry for a local file at the JANUS control point to that for an input file at control point 0; to signal this condition, DEST3 was zeroed. Now overlay subroutine B.SKFNT is called. If the return from this subroutine is with non-zero in the A register, it did not find a free FNT slot, so things are left as they are and control goes back to LPD99. If the return is with 0 in the A register, the subroutine has found an empty slot, stored its address in PS1, and put into the slot a dummy FNT entry for a file called MESSAGE, type local, at the JANUS control point. Now we can proceed to CRDA. Note that when the preceding file was terminated, the buffer was set empty, with FET pointers IN=OUT=FIRST; but the image of the new job card is in the 16 words following the end of the buffer, and FL.IMAGE is 1 to indicate that 1IR has this card in hand. Our special situation will be recognized just below CRD7F, when we find that the FET status is INITA (=3), to which it was set when the preceding file was terminated (see overlay B.RTER for the termination procedures).

134. At CRDA, the number of words that are empty in the buffer is found. (Note that LIMIT-FIRST necessarily equals BUFLG-20B). If this is less than 17, there may not be room to read in the next card, since one card, in 80-column binary reading, contains 16 words of information. In that case, subroutine WRITE is called to try to empty the buffer by writing on disk, and then we return to LPD99. If the buffer has between 17 and 177B+17 empty words, WRITE is called; if more empty words, it is not called; in either case

subroutine RES is then called to reserve the channel and connect the card reader. If the return from RES is with non-zero in the A register, the return is immediate to LPD99 as the reader has rejected a function. Otherwise, subroutine STS is called to read its status, put it in STAT, and return with it in the A register. (Note that STS is always called for the status of a card reader, but STATUS is always called for the status of a printer or punch. STATUS in fact just calls STS, and provides for a message if the printer or punch is not ready) If the busy bit of the status is 1, the reader is busy; subroutine RELCR is called to release the card reader (at CRD4D) and then return to LPD99. If the reader is not busy, the ready bit of the status is checked; if this is 0, the card reader is not busy but not ready; subroutine NRD is called to examine the situation and possibly give a message, and then go to CRD7F. (The reader is not ready; if this is because of a compare error, subroutine RCD, called at CRD7F, will not be able to read anything. But it may be because of a fail to feed or simply an empty tray. In this case, if the next card has already been read physically but not logically, FL.IMAGE will be 1 and RCD will be able to supply us with one more card.)

135. If the card reader is not busy and ready, we go to CRD7E, call subroutine CLERMES to set LSTAT to GOOD and clear out the B-display message, if any, and then enter subroutine RCD with 1 in the A register. If the card reader was not ready, NRD was called, and as it exits with 0 in the A register, RCD was entered with 0 in the A register. RCD does the following:

   1. If FL.IMAGE is 1, reset it to 0, read the card image stored after the end of the buffer into IMAGE ff., and set HAVECARD non-zero.

   Otherwise, set HAVECARD to zero.

   2. If the A register was zero on entry to RCD, exit now from the subroutine.

   3. If the A register was non-zero, read the next card to IMAGE+80 ff.

   4. Call subroutine RELCR to release the channel and card reader.

   5. Call subroutine SALARM to set the file alarm AL.CR (=33) milliseconds ahead of the current reading of the millisecond clock.

   6. Store the new card image in CM immediately after the buffer, and set FL.IMAGE to 1. Then exit.

136. So on return from RCD,HAVECARD indicates whether a card has been read logically. If not, control returns to LPD99. If so, we see whether the FET status is INITA: if not INITA, we are reading normally in the middle of a file and branch to CRD7D. But if the status is INITA, what we have just read into IMAGE ff. is the image of the job card of a new file. The INITA status was set either in overlay B.FORAG, on finding that a previously inactive reader was now ready, and after reading the first card of a new file physically; or in overlay B.RTER, on finding the 6-7-8-9 card at the

end of one file was immediately followed by another card, presumably the job card of a new file, which we had already read physically. Now we have to decode and use the job card. First column 1 is checked for 7 and 9 punches; if it has them, this is a binary card (probably one of a few extra 6-7-8-9 cards following the one that terminated the preceding file) which are not of interest, so we merely return to LPD99 hoping for better luck on the next card. The FET status is still INITA.

137. But if this would-be job card is apparently not binary, we will treat it as a job card, good or bad. We call subroutine HOL to translate the card image into display code characters, two per byte, beginning at IMAGE; to add after the last non-double-blank byte enough zero bytes, from 1 to 5, to make the total a multiple of 5; and then to put the number of CM words to which this total is equivalent into cell WC. On return from HOL, we zero D.T0 through D.T4, for use in a moment. Then copy the translated job card image into the beginning of the buffer, and set the IN pointer in the FET, and cell IN, to show that the buffer now contains this information. Then we write DEST0 through DEST4 back to the 8th word of the FET, and save DEST3, the FNT entry address, in PS1. The reason we write the word back to the FET is that the FNT address may have been found just above, between CRD and CRDA, and is not yet in the FET proper; 2TJ will be called and then overlay B.JOB, from which we shall return to LPCK in subroutine LPC, so that the 8th word of the FET will not be stored back in the normal way. The reason DEST3 is saved in PS1 as well is that overlay B.JOB will need the FNT entry address, but 2TJ will have copied the new job name into DEST0 through DEST4; which is why we must exit from B.JOB to LPCK rather than entering LPC in the normal way, which would put the job name instead of the proper information in the 8th word of the FET.

138. Now we call subroutine R.OVL to load 2TJ as an overlay in 1000B ff. of our PP memory. But as the overlay could be up to 1500B bytes long, it may extend from 1000B as far as 2477B. So before calling R.OVL, we save 2000B through 2477B in words 40B through 137B of the file buffer. This part of the buffer must be free, because only one card has so far been stored in the buffer. Next we enter 2TJ itself; but it will expect D.FIRST and D.FIRST+1 to be set up normally, so we copy their proper values out of subroutine FIRST. The two bytes are generally used as IN and OUT, but we shall not be handling this file again in the ordinary manner until we have passed through subroutine LPC, which sets them properly.

On exit from 2TJ we store the A-register in D.T0 and D.T1; if bit 17 is 1, this indicates a job card error; otherwise bits 0-14 give the time limit from the job card. Then enter overlay B.JOB.

Overlay B.JOB

First we restore cells 2000B-2477B, which were saved above before loading 2TJ, in case it might be longer than 1000B bytes.

Now fetch bit 5 of D.T0, which will be 1 if 2TJ found a job card error; shift it to the position of flag FL.JOBER, which must assume its value, and store in D.Z1.

139. If the job card is good, 2TJ will have set D.JPR, D.JECS, and D.JFL to the

priority, ECS field length, and CM field length taken from the job card. Good job card or bad, 2TJ has put the job name taken from it, expanded or contracted into five characters plus a serial number in the sixth and seventh characters, in D.EST through D.EST+3. To convert D.EST through D.EST+4 into a new first word for the FNT entry (which at present has the dummy name MESSAGE) bits 3-5 of D.EST+3 are set to 3, for type local, and bits 0-2 are set to our control point number, and the job priority is copied from D.JPR to D.EST+4. Then these five bytes are written into the first word of the FNT entry, whose address was saved above at PS1. Then the same word is altered, by zeroing bits 0-5 of D.EST+3 and setting D.EST+4 to contain INITB (=7 ; initial status with no particular meaning; it might as well be 17B = write completed, but status 7 might come in handy someday) and we copy it again to central memory as the first word of the FET.

140. Next we set up the sixth word of the FET, in D.FNT through D.FNT+4, and the seventh word, in D.FNT+5 (named G5) through D.FNT+9. These are to be:

```
VFD     12/a,12/b,12/c,12/d,12/e
VFD     1/FL.WTFNT,11/f,1/FL.EOJ,1/FL.PREAB,1/FL.JOBER,5/x,
            1/FL.IMAGE,15/g,24/h
```

where a is the field length requested on the job card, b is the ECS field length requested on the job card, c remains 0 and is the serial number of the first (control card) record of the job file, d is 1 and is the serial number of the first card in the record (job card), e is 1 and is the type number of the last card read (BCD - for explanation of types see section 144 below), f is a count that is initially 0 but does not concern us yet, x is a field initially zero but still reserved for future system use, g is the time limit from the control card, and h is the total number of cards logically read in the file so far, now=1. The flag bits will all be zero except FL.JOBER, which will be set according to the value saved above in the correct position of D.Z1; and FL.IMAGE. The latter flag was set by subroutine RCD, in the 7th word of the inchoate FET, when the job card was logically read; so here we save it from the existing G6 and restore it in the new G6 along with FL.JOBER.

After writing the 6th and 7th words back into the FET area, we read the 9th and 10th, copy LSTAT into its position in the 15th, and prepare to:

141. take the job name from D.EST through D.EST+3 and fill it into the 9th and 10th words of the FET, where it forms part of the J-display line and possible B-display line for the file, and also into the initial dayfile message at FDAZ. Then call R.DFM to put the message into the system dayfile, but not the control point dayfile, and without adding the control point name "JANUS". This is the message that looks like:

    00.00.52 XXXXX04. READ.

Finally, we restore IR2, IR3, and ZERBUF from the input register, because 2TJ has used them as D.JECS, D.JPR, and D.JFL.

Then we exit from the overlay to LPCK in subroutine LPC. LPC is not entered normally, by a branch to LPD99, because DEST0 through DEST4 contain not what is necessary in the 8th word of the FET, and the 6th, 7th, 8th, and 16th words have already been stored back into the FET.

142. So the FET has been initialized as follows:

```
VFD  42/jobname,18/INITB
VFD  60/b,60/b+c,60/b,60/b+BUFLG-20B
VFD  12/d,12/e,12/f,12/g,12/h
VFD  1/FL.WTFNT,11/j,1/FL.EOJ,1/FL.PREAB,1/FL.JOBER,5/x,1/FL.IMAGE,15/r,24/k
VFD  12/u,6/cb,6/ca,12/START,12/v,12/w
VFD  12/0HCR,12/qa,12/5555B,42/jobname,42/0
VFD  60/0,60/0,60/0,60/0
VFD  12/qb,48/0
VFD  12/m,48/y
```

where:
b is the address of the start of the buffer.
c is the number of words read from the job card and stored in the buffer.
d is the field length/100B taken from the job card.
e is the ECS field length/1000B taken from the job card.
f is the record count, initialized at 0.
g is the card count within the record, now = 1 (the job card).
h is the record type, now = 1 as the job card is BCD.
k is the total of cards read in this file, now = 1.
r is the time limit taken from the job card.
u is the JANUS control point number; useless, but the field is reserved for future system use.
cb is the second channel number, from the EST entry.
ca is the first channel number, from the EST entry.
v is the address of the FNT entry for the file.
w is the equipment connect code for the card reader.
qa is the EST ordinal of the card reader translated into two display code characters representing octal digits.
qb is the same ordinal as a binary integer.
m is whatever was last in LSTAT.
y is whatever was last in DEEP through DEEP3.
x is 0, in a field reserved for future system use.
j is the count of pre-OUTPUT message words, initially 0.
The names beginning with FL have been used illegitimately here to represent bit values 0 or 1, instead of the various powers of two. But it makes identification easier for this table:

FL.WTFNT is initially 0.
FL.EOJ is initially 0.
FL.PREAB is initially 0.
FL.JOBER is initially 1 if 2TJ found a job card error; otherwise 0.
FL.IMAGE may be 0 or 1, depending on whether the card after the job card has already been physically read into the hold.

Main Loop for Card Reading Continued

143. CRD7D is reached when any card of a file but the job card has been logically read. First 1 is added to the count of cards in this record, in bits 12-23 of the 6th word of the FET. Then 1 is added to the total card count in bits 0-23 of the 7th word of the FET. Then D.Z1 is set to point to IMAGE, i.e., column 1 of the card image. This will be a convenience at various points below. Then a check is made to see if column 1 contains a 7 and a 9 punch, ignoring for the moment all other rows; if so, a branch is made to CRD71 for a binary card; if not, continue at CRD7 for a BCD card.

144. Here the record type number, which is stored in cell RTYPE, or in bit 0-11 of the 6th word of the FET, must be explained. This field is set to 0 at the beginning of each record, and remains so, or is set to 0, whenever a normal binary card is read. It is set to some number between 2 and 80 when a card that initiates 80-column binary reading is read; such a card has 7777B in column 1 and in one of the other 79 columns, and 0000B in all other columns, and the column number, between 2 and 80 inclusive, of the second 7777B column gives the number to be stored in RTYPE. Thus in this case, RTYPE not only indicates that 80-column binary reading is being done, but specifies what sort of card (i.e., one identical with the initiating card) is needed to terminate it. When the terminating card is read, RTYPE is set to 177B, to indicate that the next card will normally be an end-of-record card. Then if the record is not ended immediately after the end of 80-column reading, there will be a mode-change message.

145. At CRD7 1IR begins dealing with a BCD card. If RTYPE contains 0, then either it is at the beginning of a new record, which is fine, or the preceding card was normal binary, which is all right but calls for a mode-change message. We branch to CRD7B where the two cases are distinguished according to whether the card count in CCT is 1. If so, we have just begun a record with a BCD card, so we go to CRD7N, set RTYPE to 1, for BCD, and continue to CRD7A. If not, there is a mode change from binary to BCD, so subroutine MODMEST is called before going to CRD7N as above.

146. If, at CRD7, RTYPE contains not 0 but 1, the preceding card was BCD like the current one, and we can go straight to CRD7A. If RTYPE contains 177B, we to to CRD7B; the preceding card terminated a series of cards read in 80-column binary, so CCT will not contain 1 (the present card is not the first in its record) and MODMEST is called to indicate a mode change from 80-column to BCD, then RTYPE is set to 1 for BCD, and then CRD7A is reached. If RTYPE contains any other number, it is presumably between 2 and 80 and indicates that this card must be read in 80-column binary. So a branch is made to CRD73 for that.

147. Thus control reaches CRD7A for a card that is to be read in BCD, or CRD73 for a card that appears to be BCD, but has to be read in 80-column binary. At CRD7A, subroutine HOL is called, which translates the card image in IMAGE through IMAGE+79 into from 1 to 40 bytes of display code information, trailing blank bytes being suppressed, beginning at cell IMAGE, and followed by 5 zero bytes. If k is the number of bytes of information, then WC has been set to contain q such that 5*q is between k+1 and k+5 inclusive. In other words, WC contains the number of CM words this card will add to the buffer, after as many trailing pairs of blanks as possible have been suppressed, and enough zero

bytes, between 1 and 5, have been added to fill out a full CM word. So we branch to CRD7P, where we put IMAGE, the starting address of the information, in XFR2, and then call subroutine XFR to copy the information to the buffer unless FL.PREAB or FL.JOBER = 1.

148. At CRD73, things are simpler. All 80 bytes, i.e., 16 CM words, beginning at IMAGE have to be transferred to the buffer. So WC is set to contain 16 and the A register to contain IMAGE, and control passes to CRD7R.

148A. At CRD7R, we have in the A register the starting address of the data to be sent to the buffer. This is now put in XFR2, and subroutine XFR is called to move the data to the buffer. However, if FL.PREAB or FL.JOBER = 1, we do not call XFR, but go straight to LPD99, as this job will be pre-aborted and there is no further reason for writing it on disk.

149. Now for a card with 7 and 9 punches in column 1, we must branch to CRD71. It may turn out to be a standard binary card, or a card to be read in 80-column binary, or an initiator or terminator of 80-column binary reading, or an end-of-record card,or an end-of-file card, or a card in no known format that causes the job to be pre-aborted.

150. If column 1 does not contain 0017B (I6789), we go to CRD72. If it does, and if RTYPE contains 0 or 1, the preceding card was normal binary or BCD, and we go to REOF to treat this card as an end-of-file. If RTYPE contains 177B, we also go to REOF, as the preceding card terminated 80-column binary reading. If RTYPE contains anything else, the current card is to be read as 80 binary columns of data unless it has 0017B in column 1 and 0000B in every other column; such a card can never be read as anything but an end-of-file. So now a check is made for 0000B in columns 2-80, and we go to REOF if so, and to CRD73, for 80-column binary reading, if not.

151. At CRD72, there is a check for 0007B (I789) in column 1. If not, we go to CRD74. If so, maybe this is end-of-record. If RTYPE contains 0, 1, or 177B, we go to REOR, just as we went to REOF in the preceding paragraph. Otherwise, the card is to be read merely as 80 binary columns, no matter what is in columns 2-80, and we go to CRD73.

152. At CRD74, there is a check for 7777B in column 1, in which case this card might be an initiator or terminator of 80-column binary reading. If not we go to CRD75. If so, we test whether one other column contains 7777B and all other columns contain 0000B. If not, we go to CRD75, but if so, the number of the second 7777B column is put in SPMODE. Now RTYPE is checked: a) if it contains 0, then either this is the first card of a record, or the preceding card was normal binary. In either case, the present card is treated as initiating 80-column binary reading, and setting RTYPE to the number of its second 7777B column, which is in SPMODE. Then, as no information is to be taken from the card, we go back to LPD99. If this is not the first card of the record (CCT does not contain 1), it represents a mode change from normal binary to 80-column, and subroutine MODMEST must be called. b) If RTYPE contains 1, the preceding card was BCD, so processing is just as in the preceding paragraph, when the preceding card was normal binary. c) If RTYPE contains 177B, the pre-

ceding card terminated 80-column binary reading, so the present card is allowed to re-initiate 80-column binary reading, but is treated as a mode change.
d) If RTYPE contains anything else, it is some number between 2 and 80; 80-column cards are already being read; and the present card cannot initiate 80-column reading, but may terminate it. So the number of the second 7777B column in this card, saved at SPMODE, is compared with the content of RTYPE. If they are not the same, then the present card is merely a data card, and we go to CRD73 to read it as such. If they are the same, the present card is a terminator; RTYPE is set to 177B, indicating that 80-column binary reading has just ended; and we go back to LPD99.

153. CRD75 is reached for a card that has 7 and 9 punches in column 1, but does not contain 0017B, 0007B, or 7777B in column1; or contains 7777B in column 1 but does not have 7777B in one other column and 0000B in the remaining 78. Now if RTYPE does not contain 0, 1, or 177B, 80-column binary is being read and we merely go to CRD73. Otherwise, the card must be normal binary if it is anything valid, so we go to CRD77, first calling subroutine MODMEST and setting RTYPE to zero, unless RTYPE already contains zero.

154. When at CRD77, the present card has to be a normal binary card to be valid. In such a card, column 1, rows 5, 6, and 7 must not be punched, and rows 12, 11, and 0 through 3 must contain a binary number between 01B and 17B, the CM-word count for the card. If so, control goes to CRD79; if not, the A register is set to RS.FOR, for a format error message, and we go to ABORT, where the job is pre-aborted. At CRD79, the word count is calculated, and put in WC. Then, if column 1 of the card had the 4-row punched, we go straight to CRD80, as this is the ignore-checksum flag. Otherwise, the sum of the contents of columns 2 through 2+5n, where n is the word count in WC, is calculated. These contents are added as if in a 12-bit accumulator with end-around carry; the total should be 0, because in punching standard binary cards, the 1's complement of the sum of all the information-bearing columns, 3 ff., was put into column 2 of each card. If the checksum is correct, control goes to CRD80; if not, RS.CKS is put in the A-register, for a checksum message and we go to ABORT to pre-abort the job.

155. If a normal binary card has an acceptable format, and either a 4-punch in column 1 or a good checksum in column 2, CRD80 is reached to check the sequence number and transfer the information to the buffer. If NUMCHY,PS contains non-zero, there has already been a sequence error message in this record, so we go to CRD90 without bothering to check the number on this card. Otherwise, we compare column 80 with the number in CCT; which is the count of cards, including the current one, read so far in the current record. If they match, we go toCRD90; if not, NUMCHY,PS is set to RS.NUM (just to make it non-zero) and with RS.NUM in the A-register, to call for a serial check message, subroutine UMES is called; then get to CRD90.

156. At CRD90, WC has already been set to contain the number of CM words of information in the card, as noted in column 1. The information begins in column 3, so IMAGE+2 is put in the A-register, and we branch to CRD7R to store the data.

158. At ABORT, a binary card has been found with unrecognizable format, or with a bad checksum in column 2 and no 4-punch in column 1. With RS.FOR or RS.CKS in the A-register to govern the choice of message, subroutine UMES is called to add a message to the pre-OUTPUT file for the job. Then FL.PREAB is set to 1, indicating that th job is to be pre-aborted, and that nothing more is to be copied from its cards to the input file on disk. Then there is a return to LPD99.

159. REOR is reached on recognizing an end-of-record card, and REOF on recognizing an end-of-file card. The only immediate difference is that for end-of-record, we can go straight to LPD99 if either of FL.PREAB and FL.JOBER is 1, indicating that nothing further is being copied to the disk file. An end-of-file card must be dealt with in any case, since it terminates the card input.

160. Now before writing an end-of-record or end-of-file on the disk file, we must wait till the FET status is not busy. To avoid hanging up other files while waiting, subroutine FILWT is used, which can call subroutine LPC while waiting. It is virtually certain that LPC will cause the card image just read to be destroyed, so the contents of the first three columns, which are all that matter for end-of-record/file, are saved first in PP memory at IMAGA,PS IMAGB,PS and IMAGC,PS. This save area amounts to 12 PP words in all, since there cannot be more than 4 card readers in use by JANUS at a time. (As this save did not have to be done for all 19 possible read, print, and punch files, it was more economical to do it in PP memory than in the FET). Having saved the columns, FILWT is called to wait until the FET is not busy. As this may involve calls to LPC, FILWT has to be called from the permanent part of 1IR, not from an overlay. Then overlay B.REOF is called. This begins by setting the e.o.r level number field in the FET to zero. Then if IMAGA,PS contains anything but 7 (I789) it must be 17B, for end-of-file, and control goes to EOF; otherwise continue for end-of-record. 1 is added to the record count in bits 24-35 of the 6th word of the FET (RCT); then bits 0-11 of the same word (RTYPE) are set to 0, the value it should have at the beginning of the next record. Also zero D.Z1, in which will accumulate the level number for this end-of-record. Now IMAGB,PS and IMAGC,PS contain copies of columns 2 and 3 of the e.o.r. card, which should contain the level number as two octal digits in Hollerith code. So these two bytes are passed through subroutine EORL. In this subroutine, if a byte is blank, the level number in D.Z1 is left unchanged and there is an exit straight to EORM. Otherwise, if the byte contains the Hollerith form of a digit between 0 and 7, EORLA is reached with the binary value of the digit in the A-register, D.Z1 is shifted three bits left; the new digit is inserted in bits 0-2 of D.Z1; and exit is normal. If a byte contains anything else, it is treated as an error but no message is given; the level number is zeroed in D.Z1 and the exit is straight to EORM.

161. At EORM; for better or worse, the level number is in D.Z1; but if it is above 17B, it is zeroed before being used. Thus the valid level number punches are 0 to 7 in column 2 with blank in column 3, or 00B to 17B in columns 2 and 3. Then zeroes are put in bits 12-23 of the 6th word of the FET (CCT), the count of cards within a record, to ready it for the next record, as well as NUMCHY,PS and MODCHY,PS. The latter cells may have been set non-zero during this record if a mode change or bad serial number was found. They are set whenever a message of the relevant type has to be written on the pre-OUTPUT file, so that only one such message will be given per record. Then the level number in D.Z1 is inserted in the FET, and with 26B, for write-end-of-record, in the A-register, we go to CEORF.

162. Arriving at EOF, because IMAGA,PS contains 17B, end-of-file can be written on the input file only if the buffer is now empty, and the last thing written on the file was end-of-record. If so, control goes straight to EOJ, where 36B, for write-end-of-file, is put in the A-register and processing continues at CEORF. Other-

wise, EOF1 is reached; it is necessary to write out a zero-level end-of-record, which will also empty the buffer if necessary; so D.Z1 is set to 0 and 1IR goes to BIN3 as if the card had been end-of-record in the first place. But first FL.EOJ is set to 1. Now after any end-of-record/file action taken on a card read file, there is a wait (calling LPC as necessary) at REOFQ until the FET shows not busy. Then the buffer will certainly be empty, and by looking at the FET status and at FL.EOJ, we can tell whether:

1. End-of-file has just been written to be followed by termination.
2. End-of-record has just been written and we must now write end-of-file.
3. End-of-record has just been written and we must look for the next record in the card input.

163. At CEORF the level number in CURFET3 has already been inserted, and 26B or 36B is in the A-register. First subroutine WTFET is called to set CURFET4 to 26B or 36B, and then CURFET through CURFET4 are copied back to the first word of the FET. Next the FNT entry address is picked up from DEST3 and subroutine PRERQ is called to begin setting up a stack request. PRERQ has read the FNT entry to MYFNT through MYFNT+14; its status and level number are set the same as those in the FET, except that if the status is 36B the level number is made 17B instead of 00B. Then the FNT entry is returned to the FNT. Then the FET address is added to the stack request; then the level number, taken from the copy of the FNT entry, is combined with the order code for write-end-of-record, and with this in the A-register subroutine DORQ is called to complete the stack request and get it inserted in the stack. Then there is a return to the permanent part of 1IR at REOFQ, to await completion.

164. At REOFQ, 1IR keeps pausing, calling LPC, and testing the FET status until it shows not busy. Then, if the status shows that it has just written end-of-file, it branches to REOFF to call overlay B.RTER and terminate the file. Otherwise, end-of-record has just been written; now if FL.EOJ is 0, this is an ordinary end-of-record and there is a branch to START to begin a new card cycle. But if FL.EOJ is 1, the last card read was a 6-7-8-9 card not preceded by a 7-8-9 card; then the flag was set and processing continued as if for a 7-8-9 card; now we must proceed as for a 6-7-8-9 card, so we go to REOFL, put 17B (I6789) in IMAGE, as if a 6-7-8-9 card had just been read, and branch to REOF.

165. At REOFF, having written end of file on the disk file, we call overlay B.RTER to terminate the file. The overlay begins by testing FL.WTFNT. This could only be 1 if this overlay had already been done for this file, and had got stuck because there was no FNT slot free for the pre-OUTPUT file for the job; if FL.WTFNT is 1, control jumps straight to RTERVY, effectively the point at which we got stuck on the last try. Otherwise, we are beginning to terminate the file for the first time. Now if FL.IMAGE is 1, the first card of the next file in the same card reader has already been read into the storage area after the buffer, so the card reader, FET and buffer will not be released. Specifically, we go directly to RTERB so as not to call overlay subroutine B.DROPBF to release the buffer this file was using. If FL.IMAGE is 0, however, the card reader is apparently empty, everything will be dropped and B.DROPBF is called before arriving at RTERB.

166. At RTERB, overlay subroutine B.AMSG is called in to write an accounting message on the system dayfile, giving the job name and the number of cards read in the file. Next the second word of the FNT entry is altered so that its fourth byte represents rewound status (RBT chain ordinal 0, byte number 0) but its second byte (current RBT pointer) and 4th byte (PRU number) are set to the ECS field length and CM field length, taken from the 6th word of the FET. However, if FL.PREAB=1, both are set =0. Next the third word of the FNT entry is altered so that bits 36-50 contain the time limit, taken from the 7th word of the FET. If FL.JOBER=1, bits 51-53 are set =6, or if FL.PREAB=1, those bits are set =7.

167. Bits 48-58 of the 7th word of the FET contain a count of the number of CM words of message to be written out on the pre-OUTPUT file for this job. If the count is 0, we go straight to RTERMA - see below. If not, the following is done before getting to RTERMA. If the job is to be pre-aborted, we will certainly not go to RTERMA right away, as there must be at least one message. Now if FL.PREAB=1, there is one more message to add - JOB PRE-ABORTED - which is done by putting the address of the message in D.T0 and calling the overlay subroutine B.WUMES. Whether the job is pre-aborted or not, we then arrive at RTERVY, and call overlay subroutine B.SKFNT to set up a dummy FNT entry for the pre-OUTPUT file. If the exit from B.SKFNT is with non-zero in the A register, a free FNT slot was not found, so FL.WTFNT is set to 1 and we return to REOFQ, from which control will return to the overlay and to RTERVY after a trip through subroutine LPC.

168. If the return from B.SKFNT is with 0 in the A register, we go to RTERVB; a free FNT slot was found and filled with an entry for an empty local file called MESSAGE at our control point, whose address has been saved at PS1. Now the name is to be changed to that of the input file being terminated, the messages written out on it, and then its control point number changed to 0. Whenever the input file is brought to a control point as a job, the system will look for a local file at control point 0 with the same name; if one is found, its name will be changed to OUTPUT and it will be assigned to the same control point, without rewinding, but positioned at the end of the first record, which will comprise the card-read-time messages written by 1IR.

169. Now to write out the messages, a minimum FET will be set up in cells RA+2 through RA+6. The messages to be written are in a mini-buffer of 200B words, whose beginning is at a-b*200B, where a is the starting address of buffer 0, found in ZERBUF, and b is the number of the present ordinary FET. Remember that FET numbers from 1 up to n, where n is the number of card readers in the system but not more than 4, are for card readers; punch and print FET's have the higher numbers. When initializing the JANUS control point, 1IQ reserved 200B words of field length for each card reader to accumulate its pre-OUTPUT messages, immediately below buffer number 0. The FIRST and OUT pointers in the special FET must point to the beginning of the 200B-word area; the LIMIT pointer must point 200B words further ahead (there will not be more than 176B words of messages in the mini-buffer); and the IN pointer must point to the beginning of the mini-buffer plus the number of words of messages in it, which number is in bits 48-58 of the 7th word of the FET (bit 59, FL.WTFNT, has been set to 0 not because it has any further logical significance, but to make the arithmetic easier.) On this basis the four pointer words are set up in RA+3 through RA+6,

as well as the FIRST pointer in subroutine FIRST. The first word of the FET, at RA+2, is supplied from MASS through MASS+4, which contains the file name MESSAGE and status 26B, as one record is about to be written. FETAD is altered from 16*(PS) to 2, the exceptional FET address.

170. The address of the FNT entry was saved at PS1 by overlay B.SKFNT; with this in the A register, subroutine PRERQ is called, which begins to set up the stack request, and copies the FNT entry into MYFNT through MYFNT+14. We set the FNT status to 26B, and copy it back to the FNT. The FET address, 2, is inserted into the stack request (as the field for this address runs from bit 6 of MYRQ+6 through bit 11 of MYRQ+5, 200B is added to the former byte.) Then the code for write-e.o.r. is put in the A register and we call subroutine DORQ to insert the code in the request and get the request inserted in the stack. Finally, we restore FETAD to its normal value.

171. At RTERMB we wait until the FET status shows completion, with a pause for storage relocation etc. after each test. At this point there is a risk of holding up all JANUS operations while waiting for this one write to be completed. The wait should be tolerably short, and should happen fairly rarely, as most jobs do not produce any pre-OUTPUT file. If it did become a serious problem, one could, instead of merely pausing after each inspection of the FET, set bit 58 of the 7th word (the bit next right of FL.WTFNT), which is otherwise unused, to 1, and return to REOFQ. Overlay B.RTER would have to check this bit right after checking FL.WTFNT, and branch to RTERMB if it was 1. CM location RA+7, which is not now used, could be an interlock for preventing us from trying to use RA+2 through RA+6 for more than one card read file at a time. In this overlay, this interlock would have to be checked immediately before calling overlay subroutine B.SKFNT, and if it was occupied, things would proceed as for the case that the return from B.SKFNT was with non-zero in the A register, no FNT slot being available at the moment.

172. When, below RTERMB, we find the write action to be completed, we change the first word of the FNT entry for the pre-OUTPUT file from MESSAGE, type local, at our control point, to the same name as the real job file, type local, control point zero. Note that if some adjustment has to be made to meet the problem suggested in the preceding paragraph, this alteration of the FNT entry has to be done before 1IR goes to RTERMA or its equivalent. The pre-OUTPUT file FNT entry must be finalized before that for the job file, as otherwise the system might find the job file, and then look for the pre-OUTPUT file without finding it, only to have the pre-OUTPUT file arrive uselessly a few microseconds later. Finally we arrive at RTERMA.

173. When we get to RTERMA, either there was no pre-OUTPUT file to write, or we have written it. If FL.PREAB=1, the priority in the FNT entry for the job file is changed to 7777B, so as to get the bad job processed as soon as possible (if FL.JOBER=1, the priority was set to this by 2TJ.) Then the FNT entry, good or bad, is changed from type local at our control point to type input at control point zero. Now if FL.IMAGE=0, the card reader was apparently empty after the end of the terminated file, and everything can be dropped. We branch to RTERMS, where the card reader is dropped from the control point (its EST ordinal is in PSTAK,PS); PSTAK,PS is zeroed to indicate that the FET is now free; the FET area is zeroed; and then we go to LPCK in subroutine LPC. The entry to LPC is not normal,

as by a branch to LPD99, since nothing need be stored back in the no-longer-active FET. The buffer has already been released at the very beginning of overlay B.RTER.

174. If FL.IMAGE is 1, however, the first card of the next job has already been read through the same card reader to the area after the buffer, the new file should be initialized with as little work as possible. The file name in the first word of the FET is changed to 9999, and the FET status to INITA (=3). The name has no significance, but the status will be seen just below CRD7F, and will cause us to initialize the new file. Skip down to RTERMV for the moment. There DEST3 is zeroed since the address that was in it pointed to what has been released from this control point. The 0 in DEST3 will be seen at CRD, and will cause overlay subroutine B.SKFNT to be called to get an FNT slot for the new file. With 0 in IN, overlay subroutine B.SETP is called to set the FET pointers so that IN = OUT = FIRST, the proper condition for the beginning of the next file. Then exit to LPD99. Subroutine LPC can be allowed to store things back in the FET normally.

175. Before going to RTERMV, however, an attempt should be made to move the buffer downward. If this is not done, and if there is a very long unbroken input of short jobs in this card reader, the buffer will never be freed (it cannot be freed between jobs as the 20B words at the end of the buffer area always hold a needed card image) and it will never be moved downward, because moving is attempted only about every 30 seconds, but every job in the series may load in less time than that. It is bad for a buffer to remain immovably assigned for a long time, because it may be rather high in the field length while buffers below it have become free; not moving the buffer downward prevents us from shrinking the field length at the control point. So here subroutine MAYMOV is called to see if the conditions for buffer-moving are fulfilled; if not, the exit from MAYMOV is with non-zero in the A-register and control goes straight to RTERMV. But if the exit is with 0 in the A-register, the conditions are apparently fulfilled, and the interlock still exists on the buffer assignment registers. So subroutine PUT is called to release the interlock, and then overlay subroutine B.MOVBF. Then RTERMV is reached. Note that in the other place in 1IR where B.MOVBF is called, subroutine FILWT is called after subroutine PUT, to wait until the FET shows not busy. But here in overlay B.RTER, it is already known to be not busy.

SUBROUTINES

Below is a list, for each subroutine, of the other subroutines it calls and the registers it destroys itself, apart from the ones that may be destroyed by the subroutines it calls. This makes it a little more convenient to work out the whole chain of calls, and the whole list of registers destroyed, that may result from a call to any one subroutine.

Subroutine	Calls	Destroys
LLWT	LDCA	D.Z1
LLRD	LDCA	D.Z1
HOL	UMES	D.Z1 through D.Z7
MODMEST	UMES	
UMES	CONVERT, B.WUMES	D.Z1, D.Z2, D.T0
CONVERT		D.Z2, D.Z3
CLERMES		D.Z1 through D.Z7
FCN	STATUS, CFR	
READY	CLERMES, STATUS, REL, SW, LPC, RES	
FILWT	PAUSE, NODAT, LPC	
RES	R.RCH, R.STB, ACC, SW	D.T2
CFR/ACC	MAYMES, REL	
STS		
STATUS	STS, MAYMES	
REL	R.DCH, PAUSE	
RELCR	REL, CFR	
NRD	MAYMES, RELCR, EXECUTE	
PAUSE	R.PAUSE	
FETCH	PAUSE	D.Z1 through D.Z5
PUT		
RCD	SALARM, RELCR, LDCA	D.Z1
PRT		D.Z1, D.Z2, D.Z4, D.Z6
PCHOLD/PGHNEW	FCN, PRT, SALARM	
SALARM		D.Z1 through D.Z5
MAYMOV	FETCH, PUT, FIRST	D.T5, D.T6
EXECUTE		D.Z1 through D.Z6
PRERQ		D.Z6 through D.T4
NODAT	WTFET, PRERQ	
DORQ	R.EREQS	D.T0 through D.T4
CIO	PAUSE, NODAT, DORQ	
WTFET		
WRITE	PAUSE, NODAT, DORQ	
MAYMES	B.MSG	D.T0
SW	EXECUTE, R.PROCES	D.Z2 through D.T5
CLR		D.Z1, D.Z2
YFR	FIRST	
MOVIM	FIRST	D.Z1, D.Z2
SAVOUT	MAKPTR	
LDCA	FIRST	
SETBR		
RUDDY	STaTUS, REL, SW, LPC, RES	

Subroutine	Calls	Destroys
XFR	FIRST, MAKPTR	D.Z1, D.Z2
IMFIL	RES, READY, FCN, PRT, REL	
FIRST		
MAKPTR	FIRST	D.T0 through D.T5
WFR	YFR	D.Z3, D.Z5
B.DROPBF	FETCH, PUT, FIRST	D.T0
B.MOVBF	MAYMOV, PUT, FIRST	D.T1, D.T2, D.T7
B.STEP		D.T0 through D.T4
B.WUMES		
B.AMSG	FIG, R.DFM	D.Z2, CT, CT+1
B.MSG	R.DFM	D.Z1, D.Z2, D.T0, D.T1
FIG		D.Z1, D.Z3, D.Z3
B.FCIA		
B.REW	WTFET	D.Z1 through D.Z5
B.SKFNT	R.RCH, R.DCH	D.Z1 through D.Z7
B.SETFT	FETCH, PUT	D.Z3 through D.T7, except D.T0, D.T6
LPC		ALL
R.OVL		D.Z1 through D.T7
R.EREQS		D.T0 through D.T7
R.DFM		D.T0 through D.T7
R.RCH		D.T0 through D.T4
R.DCH		D.T0 through D.T4
R.PAUSE		D.T0 through D.T4
R.PROCES		D.T0 through D.T4
R.STB		D.T0, D.T2

Note that any call to an overlay subroutine also involves a call to EXECUTE.

Non-Overlay Non-Hardware Subroutines

176. LLWT

This is called during the preparation of a print line image to copy the control word in KEY through KEY+4 and the image in BUF through BUF+154 (=KEY+5 through KEY+159) into the 40B CM words that lie between the end of the relevant buffer, which is BUFLG-40B words long, and the beginning of the next buffer or the end of the field length.

Entry and Exit Information

None.

177. LLRD

This is called during the preparation of a print line image to do the converse of LLWT.

Entry and Exit Information

None.

178. HOL

This is called at two points in 1IR, to translate the image of a Hollerith-punched card into display code information: at CRD7A, in the course of ordinary card reading, and below CRD7F, to translate a job card. The card has been read, one column per PP word, into IMAGE through IMAGE+79; note that for this routine to work, IMAGE must be an even address, as is apparent from the narrative at VGO below. As the information will be compressed, IMAGE ff. can be used to store the output of this subroutine as well. D.Z1, D.Z2, and D.Z7 are set to point to IMAGE: D.Z7 points to the next byte from which a Hollerith column is to be taken, D.Z2 points to the next byte into which a pair of display code characters is to be stored, and D.Z1 points to the byte in which something other than a pair of display code blanks was last stored. Of course nothing of the sort has been stored yet, but if the entire card turns out to be blank, this cell should show that one byte of information was obtained from it. D.Z6 is zeroed; this will be set non-zero if any column contains an invalid combination of punches. Then we go to VMORE to start processing a column.

179. At VMORE, the Hollerith column to which D.Z7 points is taken up; if this is 0000B, control can immediately go to VBLANK, put the display code blank in the A register, and proceed to VGO to store the display code character. Otherwise, some decoding must be done. First, the 1, 8, and 9 punches (403B) are masked off; if none of them is punched, we go to V80 with 0 in the A register; if only the 9 punch, we go to V9; if only the 8 punch, to V8, and hence to V80 with 8 in the A register; if only the 1 punch, we go to V1. If more than one of them is punched, we go to VBAD for an invalid combination. At VBAD, D.Z6 is set non-zero, to mark the error, and then control goes to VBLANK to treat the column as blank. At V9 or V1, D.Z5 is set to 9, or 1, the value the numerical punch should contribute to the BCD version of the column. Then the column is picked up again and checked for punches 2 through 7 (376B). If any of them is present, it is an invalid combination and we go to VBAD; otherwise, to VD, where we always go when the numerical punches have been evaluated.

180. When 1 and 9 punches are excluded, the allowable numerical punch combinations are: any single punch 2 through 8, or 8 and one other punch 2 through 7; i.e., 8 yes or no, and no other or one other numerical punch. The BCD values of the punches are additive; 10B for the 8 punch, and the nominal values for the others. So at V80, D.Z5 is set to contain 10B or 0, depending on whether there was an 8 punch; it has been established that there is no 1 or 9 punch; now the 2 through 7 punches are masked out, evaluated, and the resulting 0, 2, 3, 4, 5, 6, or 7 is added to D.Z5, after which we go to VD. If there turns out to be more than one punch in the range 2 through 7, this is an invalid combination and we go to VBAD.

181. At VD, the column is picked up again, and the zone punches isolated. For convenience, zone punches will be evaluated as for external BCD: 0 is worth 20B, - is worth 40B, and + is worth 60B; then the right half of table ALPH contains display code values of characters, in ascending sequence (apart from zero and blank) of the corresponding external BCD values. If + and - or +- and 0 are punched, it indicates an invalid combination and we go to VBAD. If + and 0, we go to VZ5, and if - and 0 to VZ3. In these two cases, the only valid com-

binations are the two zone punches, and nothing punched in the range 1 through 9. So at VZ5 and VZ3, control goes to VBAD if D.Z5 does not contain 0; otherwise, the A register is set directly to the display code for +0 or -0 respectively, and we go straight to VGO. If there is only a plus zone punch, control goes to VDB with 0 in the A register, and adds 3, shifts to make 60B, and adds this to D.Z5. If there is only a minus zone punch, or only a zero zone punch, or no zone punch, VDB is reached with -1, -2, or -3 in the A register, to which 3 is added and a shift performed to make 40B, 20B or 00B respectively, which is then added to D.Z5. This gives us the external BCD value; now the corresponding display code value is picked up out of the right half of table ALPH, and return to VGO is with this in the A register. Bowever, if the value in table ALPH is 00B, we go to VESCAPE instead; as things stand, this is just another branch to VBLANK, and means that an 8-6 punch is read as a blank.

182. Because there are 64 different valid punch combinations (actually 66, but plus-8-2 and minus-8-2 are universally allowed to count as plus-0 and minus-0) and only 63 possible display code characters (since 00B has to be reserved as a trailing blank), two punch combinations must either be treated as equivalent (blank and 8-6 the way table ALPH now stands) or else each of two or more input characters translated into a pair of characters in memory. This could be done by giving each of these special input characters the value 00B in table ALPH. On branching to VESCAPE, instead of just going to VBLANK, we would then look at the original column or at D.Z5, to see which special character was present, and accordingly choose two display code characters to add to the translated card image. However, this would require an adjustment of what is done at VGO and VGOA, as it would no longer be possible to tell from the mere address of a column in the original card image whether the corresponding display code character was to go into the left or the right half of a byte in the translated card image.

183. At VGO, the display code value of a Hollerith column is in the A register, and it is saved in D.Z3. Now 1 is added to D.Z7 to make it point to the next Hollerith column. If this is an odd address, we go to VGOA to put the display code character in the left half of the byte in the translated card image to which D.Z2 points. If it is an even address, the display code character is inserted into the right half of the byte D.Z2 points to. As the byte is full, we compare it with two blanks; if it is not two blanks, its address goes in D.Z1, which is to contain the address of the rightmost non-double-blank byte. Then 1 is added to D.Z2. Finally, there is a return to VMORE for the next Hollerith column unless a display code character has just been inserted in the right half of IMAGE+39, whereupon the whole card has been translated.

184. At HOLA, the address of the last non-double-blank byte, or the address of the first byte if the whole card was blank, is in D.Z1; it is saved at HOLB+1 because subroutine UMES may be called, and would destroy D.Z1. Later 1 is added to this address, and five zero bytes are read into that cell and the following 4, to make a terminator for the card image. Now if D.Z6 contains 0, control goes straight to HOLB. Otherwise, at least one column on the card contained an invalid combination of punches, and subroutine UMES is called, with RS.HOL in the A register to select a HOLLERITH CHECK message for the pre-OUTPUT file of the job. At HOLB, we find the number of bytes in the translated card image up to and including the last non-double-blank byte, plus 1 because the image must end with at least one zero byte. Then the result is divided by 5, and the

quotient, increased by 1 if there is any remainder, is put in WC. This is the count of CM words, including from 1 to 5 zero bytes in the last word, that must be transferred to the buffer as the card image. The five zero bytes are read in as explained already, and an exit is taken from subroutine HOL.

Entry Information

The Hollerith card image in IMAGE through IMAGE+79.

Exit Information

The display code card image, with trailing blanks replaced by 1 to 5 zero bytes, in IMAGE through IMAGE+5n-1, where n is the CM word count in WC.

185. MODMEST

This subroutine is called whenever a mode change is found within a record of cards being read in; i.e., when a normal binary or BCD card follows a card not of the same type, or when 80-column binary reading is initiated other than at the beginning of the record. If cell MODCHY,PS is non-zero, a mode change message has already been added to the pre-OUTPUT file on account of this record, and an exit is taken from MODMEST without adding another message. Otherwise, MODCHY,PS is set non-zero, and then, with RS.MOD in the A register to select the MODE CHANGE message, subroutine UMES is called. MODCHY,PS is zeroed at the end of each record.

Entry Information

Zero or non-zero in MODCHY,PS.

Exit Information

Non-zero in MODCHY,PS.

186. UMES

This subroutine is called with some number between 1 and 6 (RS.MOD and RS.END) in the A register, to add the corresponding message to the pre-OUTPUT file for the job that is being read in from a card reader at the current FET. The message number is saved in D.T0; then overlay B.UMES is called. The overlay includes the messages; it gets the message number from D.T0 and finds the starting address of the corresponding message and saves it in D.Z1. Then it copies the message into MESSAGE through MESSAGE+5, immediately before the counts. Next it calls subroutine CONVERT twice; one to convert the number of the current record (in RCT, which corresponds to bits 24-35 of the 6th word of the FET) to decimal and insert it after RC. in the total message; and again to convert the number of the current card within its record (in CCT, which corresponds to bits 12-23 of the 6th word of the FET) to decimal and insert it after CD. in the total message. Then the total message is something like

HOLL.CHECK RC.00001,CD.0012

which would mean that an invalid Hollerith punch was found in at least one column of the 12th card of the 2nd record of the job. This makes a message of 3 CM words, of which the last byte is 0000B, the format required by subroutine WUMES. The starting address of the message, MESSAGE, is put in D.T0, and overlay subroutine B.WUMES is called to add it to the pre-OUTPUT file for the job. Then control returns to exit from subroutine UMES in the permanent part of 1IR.

Entry Information

The message number in the A register.

Exit Information

None.

187. CONVERT

This is called twice in the overlay part of subroutine UMES (B.UMES) to convert the number in the A register (which may range up to 9999, although in fact no number greater than 2**12-1 could be fed to CONVERT, as the count of records in a file, or cards in a record, would loop back to 0 after this maximum) to four decimal digits, each between 00B and 11B, and add them into the 4-character field in the byte D.Z1 points to and the one following it. These bytes already contain display code 0000.

Entry Information

The number to be converted in the A register, and the pointer in D.Z1.

Exit Information

None.

188. CLERMES

This subroutine is called whenever (with a few exceptions) a unit is found to be ready and not busy, in order to clear any B-display message there may be in the FET for the file, and to set LSTAT to GOOD if necessary. CLERMES is also called to do these things when the operator types in OK to end the display of a PM-line from a print file.

189. If LSTAT contains GOOD, it can be assumed there is no display message in the FET, and an exit taken immediately. Otherwise, LSTAT is set to GOOD, and then we zero bits 0-35 of the 10th, and all of the 11th through 14th words of the FET. This cancels any B-display message for the file, and cuts the J-display line back to the equipment type, EST ordinal, and job name in the 9th word of the FET and bits 36-59 of the 10th word.

Entry Information

None.

Exit Information

LSTAT contains GOOD.

190. READY

This subroutine is called at various points in the main loop for printing and punching, to wait until the unit is ready, or to call for switching to a different unit if requested by a type-in, without holding up other files. This means that subroutine LPC has to be called; and whenever LPC is called from a subroutine, the return address has to be preserved beforehand and restored afterwards. (indeed if the subroutine that calls LPC were itself called from a subroutine, not from the main loop, then the return addresses for both the inner and the outer subroutines would have to be preserved and restored. This is just an extension of what LPC itself does.)

191. We get 0001B (for a printer) or 0101B (for a punch) from STATCON,TYPE, alter it to 0003B or 0103B, and save it. Then we call subroutine STATUS, from which we return with the status byte from the unit in the A register. Now we mask it with 0003B or 0103B, invert the rightmost bit, and branch to READYX if the result is 0. This means that the unit is ready and not busy, and is not in fail-to-feed status if it is a punch. At READYX, for printers but not punches, we call subroutine CLERMES to set LSTAT to GOOD and clear away any B-display message; then exit from READY.

192. If the status is not satisfactory, we call subroutine REL to release the channel, 6681, and equipment. Now we call subroutine SW with 2 (HS.BUSY) in the A register. The 2 means that SW will do nothing if the same bit of the status byte is 1, indicating busy; presumably it is just a question of waiting till the unit is not busy again. But if the trouble with the status was rather a non-ready or fail-to-feed status, SW will try to switch to a different unit of the same type if this has been called for by an operator type-in.

193. Now LPC is called to give other files a chance; then subroutine RES is called so that the status test can be repeated. If the return from RES is with 0 in the A register, we go back to READYB to repeat the whole cycle. If not, however, the unit has rejected a function, and everything has been released; now we return to READYC, as if we had found unsatisfactory status and called REL. But this time the A register contains 1 (the return from RES is always with either 0 or 1 in the A register); entering SW with this will cause it to try to switch units even if the status is busy. We do this because if the unit is rejecting functions there is no way of reading a new status byte from it, so there is no point in considering whether it has busy status or not.

194. Note that DEEP is at LSTAT+1, and is saved in bits 36-47 of the 16th word of the FET, and restored from there, by subroutine LPC.

Entry Information

None, but the unit must be connected.

Exit Information

None, but the unit is still connected.

FILWT

195. Just as READY waits economically for a printer or punch to be ready, FILWT waits econimically for the current FET to show not busy status. If FILWT was entered with 0 in the A register, this is all it does. Otherwise, after the FET becomes not busy, the original contents of the A register are reloaded, and then subroutine NODAT is called to insert the status in the FET and the FNT entry, and to begin setting up a stack request to carry out the action specified by the status.

196. As subroutine LPC is called while waiting for the FET to be not busy, it is necessary to preserve the return address of FILWT, and the number that was in the A register on entry to it. These are saved in DEEP and DEEP3 respectively, which are preserved in bits 36-47 and 0-11 of the 16th word of the FET.

Entry Information

The A register contains 0 if FILWT is merely to wait, or some active file status (40B and 240B are the ones actually used at present) for an action to be initiated when the FET shows not busy.

Exit Information

If the A register contained non-zero on entry, this value has become the status in the FET and FNT, and subroutine NODAT has begun setting up a stack request, and has put a copy of the FNT entry in MYFNT through MYFNT+14.

PAUSE

197. This is called whenever there is a risk of being held up by a storage move:

1. In subroutine FILWT, and at REOFQ, while waiting for an FET to be not busy.
2. In subroutine REL, which is called when an equipment is not ready.
3. In subroutine FETCH, which is called to secure access to the buffer assignment registers. 1IR may be locked out by 1IQ while the latter expands or contracts storage at the JANUS control point.

4. In subroutines CIO and WRITE, for reading and writing, respectively, immediately after the request has been put in the stack, or after it has been decided not to make a new request. The only necessity for PAUSE here is probably when no new request is made because the status is already read or write incomplete. Processing may be held up by lack of space or information in the buffer until the earliest request is complete, while the request may be held up until a storage move has been done.

5. In overlay B.RTER, while waiting for the completion of writing a pre-OUTPUT file for a job just read in.

198. Subroutine PAUSE merely calls subroutine R.PAUSE, and then resets D.RA and D.FL; if the error flag is 0, exit from PAUSE; if F.ERPP, drop the BP; if anything else, set switches LPCM and LPCN to prevent future calls to B.FNT and B.FORAG, so that 1IR will drop after completing current files.

Entry and Exit Information

None.

199. FETCH

This secures access for 1IR to the buffer assignment registers in control point +77B through 104B, locks out 1IQ from them, and copies the registers to KEY through KEY+4 and BUFAS through BUFAS+24. For a full explanation, see the description of subroutine FETCH in program 1IQ.

Entry Information

None.

Exit Information

The buffer assignment registers in KEY through BUFAS+24. The A register contains address CP+105B, a fact made use of in overlays B.FNT and B.FORAG.

200. PUT

This returns the buffer assignment registers to control point +77B through 104B, after updating KEY+2, and releases the interlock so that 1IQ can get at them. For a full explanation, see the description of subroutine PUT in program 1IQ.

Entry Information

The buffer assignment registers in KEY through BUFAS+24. KEY+2 may need updating, so that it will be at least equal to KEY+3.

Exit Information

The A register is non-zero, a fact used in subroutine MAYMOV.

201. SALARM

This subroutine sets the alarm for the current FET to the present value of the millisecond clock plus the number in the A register on entry. It is called:

1. In subroutine RCD, just after reading a card, with AL.CR (=33) in the A register, because the next card cannot be read until 50 msec. from now.
2. In subroutine PCHOLD/PCHNEW, just after subroutine PRT has been called to punch a card, with AL.CP (=200) in the A register, because the next card cannot be punched until 240 msec. from now.
3. Below SPC52, after subroutine PRT has been called to print a line, with AL.LP (=40) in the A register, because the next line cannot be punched until 60 msec. from now.
4. At BACKA, with the maximum useful value of 740B in the A register, while waiting for 1IU to complete a backspace.
5. In overlay B.FNT, when setting up a file to print or punch is begun, with 0 in the A register, as the equipment may be ready now.

Entry Information

In the A register, the number of milliseconds ahead by which the alarm is to be set.

Exit Information

The alarm duly set.

MAYMOV

202. This subroutine is called, about every 30 seconds for any file, to see whether there should be a switch from the buffer now in use to one lower down in memory. It is also called in overlay B.RTER when a card reader file has just been terminated, but the first card of the next file is already in memory. If such a buffer move is not desirable and possible, nothing special has to be done. If it is, on exit from MAYMOV the 1IR PP has the interlock and access to the buffer assignment registers, and this must be dropped by subroutine PUT as soon as possible. In that case, the number of the current buffer is in D.T5, and the number of the buffer into which its contents could move is in D.T6. (Buffers should be used as low in memory as possible, so that inactive buffers will be concentrated at the upper end of the field length, and program 1IQ may be able to shrink the field length when more than one buffer is unused. See the description of program 1IQ, at KC.)

203. MAYMOV begins by calculating the number of the current buffer, using the starting address of buffer number 0 in ZERBUF, and saves it in D.T5 (permanently) and D.T6 (temporarily). If this is less than 2, exit immediately with the A register non-zero, indicating nothing doing, because we could not move down from buffer 0, and there would be no point in moving from buffer 1 to buffer 0, as we do not deliberately get rid of the only free buffer in the field length. Otherwise,

subroutine FETCH is called to get access to the buffer assignment registers and copy them to KEY through KEY+4 and BUFAS through BUFAS+25. Now if the count of all buffers in the field length, in KEY+1, equals the count of buffers assigned, in KEY+3, there is no free buffer to move to, so control goes to MAYMOVB and calls subroutine PUT to release the interlock on the buffer assignment registers. PUT exits with non-zero in the A register, and control exits from MAYMOV with the same non-zero to indicate nothing doing. Otherwise, it goes to MAYMOVA. Let the number of buffers in the field length be n; then scan back from BUFAS+n-1 to BUFAS+k, where k is the number of our present buffer; if any of these cells contain non-zero, ours is not the highest buffer in use, and there is no need to try to move it, so go to MAYMOVB as above. Otherwise, scan BUFAS through BUFAS+k-2; if none of these is zero, there is no free buffer below ours and control goes to MAYMOVB as above. Otherwise, leaving the number of the lowest free buffer in D.T6, exit from MAYMOV with 0 in the A register, indicating that the move can be made.

Entry Information

None.

Exit Information

If the move cannot be made, none except the non-zero in the A register. If it can, the A register contains 0, the number of the present buffer is in D.T5 and the number of the buffer into which the move can be made is in D.T6; we have the interlock for the buffer assignment registers and so must not drop the project without calling subroutine PUT.

EXECUTE

204. The macro EXECUTE k is expanded to LDN k; RJM EXECUTE. So the subroutine is entered with the number of an overlay (B.something) in the A register. The directory of overlay is in RA+10B through RA+17B. The first word in the table contains the addresses of overlays 1 to 5, the second word those of overlays 6 to 10, and so on. Now we multiply k by .144B and discard the fractional part of the result; as k is always below 50B this gives the integral part of the quotient (k-1)/5, q, which is stored in D.Z2. Adding RA+10B to this will give the address of the wanted word in the index. Then we subtract 5q from k and add 1, which gives the byte number, 1 to 5, within the word. Then fetch the byte from the index, and in turn fetch the word it points to, at RA+m, which has the form:

```
VFD   12/a,12/b,12/c,24/0
```

where a is the length of the overlay in CM words, not counting this first word; b is the starting address of the area in this PP into which it is to be loaded, somewhere between 1000B and 1777B; and c is 0 if the overlay is to be treated as a subroutine and non-zero otherwise. So from RA+m+1 through RA+m+a a CM words are copied into cells b through b+5a-1 in our PP. Then c (in D.Z3) is checked. If it is non-zero, a branch is taken to the start of the overlay,

at b. If c is zero, the return address at EXECUTE is copied into the second cell of the overlay, at b+1, the first cell being assumed to be LJM; then branch to b+2 in the overlay, so that if the overlay exits by branching to its first cell at b, this will return control to the point in 1IR immediately after the call to EXECUTE.

Entry Information

The overlay number in the A register.

Exit Information

The overlay set up and entered.

PRERQ

205. This is called, with the address of the FNT entry for the current file in the A register to copy that entry to MYFNT through MYFNT+14, and to begin setting up a stack request in MYRQ through MYRQ+9, by:

1. putting the address of the FNT entry in bits 36-47 of the first CM word.
2. putting the physical unit number, or 1 if the file is empty, into bits 0-5 of the first CM word, setting bits 6-11 to 0.
3. putting the FIRST and LIMIT pointers into bits 24-41 and 0-17 of the second CM word, and setting the rest of the word to 0.
4. setting bits 48-59 and 24-35 of the first CM word to 0.
5. leaving bits 12-23 of the first CM word to be set to the order code by subroutine DORQ, just before it calls R.EREQS.

206. All that needs to be explained is how to get the physical unit number. Bits 36-47 of the second word of the FNT entry contain k, the number of the first RBT word pair for the file. If this is 0, control goes to PRERQA to set the PU number arbitrarily to 1. Otherwise, q is gotten from bits 0-11 of CM location P.RBR. 100B*q is the length of central memory. The first RBT word pair for the file is at 100B*q-2k and 100B*q-2k+1, and the first word of the pair is read. Bits 39-47 of this CM word contain r, the RBR ordinal. The starting address of RBRs is in bits 36-53 of the word at CM location P.RBR; call it s. Then s+38r is the address of the first word of the RBR for the file; bits 42-47 of this word contain the physical unit number.

Entry Information

FNT entry address in the A register.

Exit Information

The FNT entry has been copied into MYFNT through MYFNT+14; the stack request has been partly set up in MYRQ through MYRQ+9 as explained above; the A register contains 0 (this is used in overlay B.TER.)

207. NODAT

This subroutine is called from subroutines FILWT, WRITE and CIO, to set up a stack request and to set the FNT entry and the FET to a new active status. It is entered with the status code in the A register; this is saved and also filled into the FET using subroutine WTFET. Then the address of the FNT entry is loaded from DEST3 to the A register, and subroutine PRERQ is called to fetch the FNT entry to MYFNT through MYFNT+14 and set up most of the stack request. Then the FET address is inserted in bits 42-59 of the second CM word of the stack request. Then the level number field in the FNT entry is zeroed, and the entry is written back to central memory. Then an exit is taken.

Entry Information

The new active status code in the A register.

Exit Information

The stack request has been partially set up in MYRQ through MYRQ+9; in fact it is complete except for the function code that will be put into MYRQ+3 at the beginning of subroutine DORQ, and the control point number, which will be inserted in bits 6-11 of the first CM word of the request by subroutine R.EREQS.

208. DORQ

This subroutine is called, with a stack request function code in the A register, when a stack request has been set up in MYRQ through MYRQ+9 all but the function code, and is to be inserted in the request stack.

The function code (including level number in the case of write-end-of-record) is inserted in bits 12-23 of the first CM word of the request; then (as subroutine R.EREQS requires), D.T1 through D.T4 are zeroed and D.T0 is set to point to the first byte of the stack request. Then R.EREQS is called, on return from which it is known that the request has been inserted in the stack, so there is an exit from DORQ.

Entry Information

The function code (including level number if write-end-of-record) in the A register; the request, lacking only this and the control point number, in MYRQ ff.

Exit Information

None.

CIO

209. This subroutine is called to do all the reading from disk for print and punch files. A check is made to see whether the FET showed busy the last time its first word awas read into CURFET through CURFET4. If so, there is no point in making up a stack request, because stack processor will already have read

as much as possible into the buffer, and no more than one card or line could have been removed since the first word of the FET was last read. Or if the FET shows end-of-information, an exit is taken from CIO without doing anything, because an attempt to read would be useless. If there has been no exit from CIO for any of these reasons, 12B is put in the A register and subroutine NODAT called to set that status (read incomplete) in the FET and the FNT entry, and to set up a stack request to read the file, all but the function code in bits 12-23 of the first CM word of the stack request. Finally, with the function code for reading in the A register, subroutine DORQ is called to complete the stack request and get it inserted in the stack. Then exit from CIO.

210. Note that whenever there is an exit from CIO, subroutine PAUSE is called first to pause for storage relocation. We may be waiting for stack processor to do a read for us, while stack processor is waiting until a storage relocation is done.

Entry Information

None.

Exit Information

None.

WTFET

211. This subroutine is called with some FET status in the A register, to store that status in CURFET4 and then insert it in the FET by copying CURFET through CURFET4 into the first word of the FET.

Entry Information

In the A register, the new FET status.

Exit Information

None.

WRITE

212. This subroutine does for card read files what CIO does for punch and print files, i.e., causes disk transmission, in this case writing. First CURFET4 is examined to see whether, the last time we read the first word of the FET, it showed busy status. If so, exit after pausing for storage relocation. There is no point in making a further request to write, because stack processor will have tried to write as much as possible from the buffer anyway, and more than 16 CM words could not possibly have been put into the buffer since we last called it. But there should be a pause, because stack processor might be waiting for a storage relocation to be completed.

213. Otherwise, we set to 0 the level number field in CURFET3; this will be copied into the FET itself when NODAT calls WTFET in a moment, and NODAT always sets the corresponding field of the FNT entry zero. Note that the only time this field in the FET and the FNT entry should be nonzero is when an end-of-record/file card has just been read; at that time, not subroutine WRITE but overlay B.REOF is called. With 16B in the A register, for write incomplete, we call subroutine NODAT to set that status in the FET and the FNT entry, and to set up a stack request, all but the function code in bits 12-23 of the first CM word, to write the file. Finally, with the function code for writing in the A register, we call subroutine DORQ to complete the stack request and get it inserted in the stack. Then we exit from WRITE, this time without calling PAUSE; if the write action just called for is held up because of a storage move, at any rate WRITE will be called again soon; we shall see the busy status then and exit with a PAUSE call.

Entry and Exit Information

None.

MAYMES

214. This subroutine is called to produce one or two appropriate messages whenever an equipment gives a bad status, unless that status has already been recognized and signalled, and no different status has been recognized in the meantime. The calling sequence is:

```
LDC     x*10000B+z*100B+y
RJM     MAYMES
```

If LSTAT contains status x, nothing needs to be done and we exit immediately. Otherwise, LSTAT is set to contain x; VFD 6/z,6/y is stored in D.TO, and we call overlay subroutine B.MSG to produce the messages. If only one message is wanted, z will be zero. On return from B.MSG, we exit from MAYMES. Note that the values x may have are not hardware statuses, but arbitrary numbers from 1 to 5 named GOOD, UNREADY, REJFNC, XMSNPE, and WROGER.

Entry Information

The calling sequence, and LSTAT.

Exit Information

LSTAT updated.

SW

215. This subroutine is called in three ways:

1. In subroutines READY and RUDDY, while waiting for a printer or punch to be ready and not busy, or ready and memory not busy, SW is called with HS.BUSY (=2) or HS.MEM (=100B) in the A register.

2. In subroutines READY and RUDDY, SW is called with 1 in the A register, after getting a non-zero (i.e. 1) response from subroutine RES, showing that the punch or printer has rejected a function.

3. In subroutine RES itself, SW is called with 0 in the A register, after a punch or printer has given a non-zero response from subroutine ACC, showing we were unable to connect it.

216. We want to treat cases (2.) and (3.) in the preceding paragraph alike; so on entering SW we zero the rightmost bit of the A register. Then we match the remaining content of the A register with the content of location STAT, which is where the printer or punch status byte is stored whenever it is read. If they have a 1 bit in common, exit from SW without doing anything. When we call SW with HS.BUSY or HS.MEM in the A register, it is because the equipment status either shows the corresponding bit on, or shows the ready bit off, or both. If the HS.BUSY or HS.MEM bit is on, probably we have only to wait for it to go off in the normal way, so there is no need to think about switching, and we exit from SW. If HS.BUSY or HS.MEM is off, but the ready bit is also off, something is wrong with the equipment and we want to continue. If HS.BUSY or HS.MEM is on, but ready is off, we exit though it would be reasonable to continue. However, we assume that HS.BUSY or HS.MEM will shortly go off, and on a future trip through SW we will have the situation of HS.BUSY or HS.MEM off and ready off, and at that time will try to switch.

When we call SW with 0 or 1 in the A register, we want to try to switch units no matter what the status.

217. If we do not exit from SW as described in the preceding section, we check FL.SWCH and exit if it is 0. If a type-in "/SWuu." has been processed at the beginning of the main loop, or in subroutine PRE called by LPC, this bit will be 1. If so, we get the two letters for the proper equipment type from the 9th word of the FET, and request monitor for another of this type. If the response is zero, no other is available, so we exit from SW. Otherwise, call overlay B.SWICH. There would have been no logical difficulty about moving more of this routine from the permanent part of 1IR into the overlay, as there is no question of calling LPC. But SW may be called very frequently while one equipment is disabled and the file is waiting for another one to become free, and it seems better not to spend the time on repeatedly loading the overlay until there is a free equipment.

218. When we enter B.SWICH, the EST ordinal of the new equipment is in D.TO. First we zero FL.SWCH, as its request is now being obeyed. Then read the EST entry for the equipment, and copy its connect code and channel number(s) into DEST4 and DEST1. We save the old EST ordinal from PSTAK,PS and replace it with the new one. Then set the old unit to off status in the EST (bit 23 becomes 1) and ask monitor to release it from this control point. Then we replace the old EST ordinal by the new one at both positions in the FET where it is recorded - in binary in bits 48-59 of the 15th word of the FET, and as two octal digits in display code, in bits 36-47 of the 9th word. Finally, if TYPE indicates a print file, we set FL.PAGE to 1 to ensure that the continuation of this file will start on a fresh page on the new printer. Then return to the permanent part of 1IR, to exit from SW.

Entry and Exit Information

None.

CLR

219. This subroutine is called to zero a card image area in IMAGE through IMAGE+79; whenever preparing to punch out a card (except for the first card of a file, with the job name visible on it); and also in overlay B.TER just to get a large zeroed area to copy into the FET area in central memory.

Entry Information

None.

Exit Information

The A register contains 0, a fact used in overlay B.EOPCH.

YFR

220. This is called for a print or punch file with a PP address x in the A register. If the buffer is empty, it exits with 0 in the A register; otherwise, it reads the next CM word from the buffer into x through x+4, updates the pointer in OUT, and exits with non-zero in the A register.

Entry Information

The address of the destination in PP memory, in the A register.

Exit Information

Zero in the A register if the buffer was already empty; otherwise the A register is non-zero and the PP OUT pointer has been updated.

MOVIM

221. This is called to save the image of a card about to be punched, or to recover the card image most recently created or its immediate predecessor. The image area in PP memory is IMAGE through IMAGE+79. The save area in CM is either x+BUFLG-40B through x+BUFLG-21B, or x+BUFLG-20B through x+BUFLG-1B, where x is the address of the first word of the buffer. A buffer of BUFLG words is provided for every file, but in a punch file the LIMIT pointer is set only to FIRST+BUFLG-40B, so that these two areas are free for image saving. They are used alternately, and FL.FLIP is inverted to alternate them, after each card has been physically punched and its predecessor successfully checked.

222. MOVIM is entered with 777776B, or 000000B, or 000001B in the A register. Only bits 17 and 0 are significant. If bit 17 is 1, the image is written to the free save area, i.e., the one not used the last time a card image was saved; the choice of save area is specified by bit 0 being 0. If bit 17 is 0, we should read back from a save area into IMAGE through IMAGE+79; if bit 0 is 0, this is the save area most recently used, i.e., MOVIM fetches the image that was most recently constructed for punching, and stored by a call to MOVIM with 777776B in the A register. But if bit 0 is 1, MOVIM fetches the card image from the other save area, i.e. the second-most-recently constructed card image, which is the image of the card to be checked for compare error. Remember that when a card is found to give a bad punch status, the next card has already been punched, so both have to be repunched.

223. When MOVIM is entered, the A register is saved (bit 0 is the only interesting thing saved) and then a central memory read or central memory write instruction selected according as bit 17 is 1 or 0. Then the save area is selected; this begins at LIMIT if bit 0 of the A register on entry is the same as FL.FLIP, and LIMIT+20B if the two bits differ. Then the image is read/written from/to this area, and an exit is taken from MOVIM.

Entry Information

In the A register, bit 17 is 1 for saving and 0 for recovering a card image; bit 0 is 0 for saving a card image not yet punched, or for recovering such an image or the image of the card last punched before its predecessor has been successfully punched; bit 0 is 1 for recovering the image of the preceding card.

Exit Information

None.

SAVOUT

224. This subroutine sets the OUT pointer in the FET according to the pointer in PP memory. Calling it has the effect of making irrevocable the removal of words from the CM buffer.

Entry and Exit Information

None.

224a. LDCA

This subroutine puts the absolute address of the last+1 word of the buffer into the A-register; it is equivalent to what LDCA D.LIMIT would do if D.LIMIT and D.LIMIT+1 were used in the normal way. A special subroutine is provided for this "LDCA" because it is so often called to get at the card or print line image that follows the buffer. First subroutine FIRST is called, to put the absolute address of the beginning of the buffer in the A register; then the length of the buffer, taken from the table at LIMIT, is added.

Entry Information

None.

Exit Information

The absolute address of the last+1 word of the current file buffer, in the A register.

224b. SETBR

This is called at various points in the construction of a print line image, to put the address of the instruction next following the RJM SETBR instruction into the second byte of the control word for the line image. Then we return to the

instruction following RJM SETBR. This is done so that if the construction is interrupted by a branch to LPD99, then the next time this file is taken up, subroutine LPC will exit to START, from which we shall branch to LPR; then we shall come to LPRQ, and because FL.LINE=1, branch to the address given in the second byte of the control word, at KEY+1.

Entry and Exit Information

None.

224c. XFR

This is called to add to the current contents of the circular buffer a number of CM words given in WC, which begin at the address in XFR2. The IN pointers in the PP and the FET are adjusted accordingly. This is called most commonly during card reading, to store a new card image in the buffer. It is also called in overlay B.LPEOR, during printing, to add a zero word after the last word in a logical record, if this word does not end in a zero byte.

First we subtract the content of IN from the length of the buffer, to find the number of empty words at the end of the buffer. Then we store in D.Z1 this number, or the content of WC, whichever is less; i.e., the number of CM words to be stored beginning where IN points. Then, at XFR1, branch straight to XFR4 if WC contains 0, as there are no words to be moved and we have effectively finished. Probably this never happens. Next store the difference between WC and D.Z1 in D.Z2; this is the number of words, if any, that will have to be stored at the beginning of the buffer after filling the vacant space at the end. Now call subroutine FIRST and add the IN pointer to the A register, giving the absolute address of the word in the buffer to which IN points; then store (D.Z1) words, taken from the area to which XFR2 points, in the buffer.

Note that if D.Z1 contained 0 this would be a desastrous proceeding, so that we depend on JANUS and the stack processor never to leave the IN pointer equal, in effect, to the LIMIT pointer, but to reset it to the FIRST pointer in every such case.

After moving information to the empty space at the end of the buffer, we check D.Z2, which contains the number of CM words to be moved to the beginning of the buffer; if this is 0, we are finished and branch to XFR4. Otherwise, take the starting address of the material just copied from XFR2; add five times the number of CM words so far, and store the result in XFR3 as the starting address of the rest of the material to be copied. D.Z2 already contains the number of CM words in this remainder, so we call FIRST to get the beginning address of the buffer, and copy the rest of the information to the buffer.

At XFR4, WC still contains the total number of CM words moved, and we add it to the IN pointer in the PP. If this is now equal to or greater than the length of the buffer, it must be reduced by the length of the buffer; hence it will never point to LIMIT. Then call MAKPTR to make an FET pointer word showing the true relative value of IN, store it in the third (IN) word of the FET, and exit.

Entry Information

XFR2 has been set to point to the first byte of the first word to be copied to the buffer, and WC contains the number of CM words to be moved.

Exit Information

IN and the the FET IN pointer have been updated, and WC remains unchanged.

224d. FIRST

This subroutine does what LDCA D.FIRST would do, if D.FIRST and D.FIRST+1 were used in the normal way. We set it up as a subroutine to free those two direct cells for another use. Calling the subroutine does not take much longer than the LDCA procedure, and the only inconvenience is having to set up the instruction at FIRSTA, and to extract the pointer from there when it is needed as a relative rather than an absolute address. But all instances of those actions are infrequently carried out, except for setting FIRSTA each time control switches from one file to another.

Entry Information

None.

Exit Information

The absolute address of the first word of the buffer, in the A register.

224e. MAKPTR

This is called whenever we have to reset the FET IN or OUT pointer. On entry, the content of PP pointer IN or OUT is in the A register. This is saved; then we call subroutine FIRST to get the absolute address of the start of the buffer in the A register; subtracting 100B times the content of D.RA gives the relative address, which the FIRST pointer of the FET contains. We add the content of IN or OUT to this, and store the result in the right two bytes of D.T0 through D.T4, which was previously zeroed. Then D.T0 through D.T4 is the wanted pointer word. On return from MAKPTR, we shall want to write this word into the FET; so before leaving, we put the address of the first word of the FET in the A register.

Entry Information

In the A register, the amount by which the new pointer is to exceed the address in the FIRST pointer of the FET.

Exit Information

D.T0 through D.T4 hold the new pointer, and the A register contains the address of the FIRST pointer in the FET.

224f. IMFIL

This is called in two situations when printing a file on a 512 printer: when beginning the file, and whenever we resume printing after stopping because bit 10 or bit 2 of the printer status byte was 1. The subroutine fills the "image buffer" of the printer, thus defining, for each position on the printer train, the character code as transmitted by the PP that is to correspond to it.

First we store the return address in DEEP1, as we shall call subroutine READY, which calls LPC, and may also call LPC directly, and in either case other files may be worked on and may cause this subroutine to be called. Note that READY saves its return address in DEEP; so there is no conflict. Then we call subroutine RES to reserve the channel, select the 6681, and connect the printer. If the return is with 0 in the A register, all is well and we go ahead; otherwise the connect was rejected and everything has been released already; so we call LPC to allow other files to go ahead and then return to the RES call at IMFILE. When RES has been successful, we call subroutine READY to wait until the printer is ready and not busy. Then put function code 12B in the A register and call subroutine FCN to send it to the printer. This function will cause the next print line image sent to the printer to be copied into its print image buffer, rather than being printed.

If the return from FCN is with non-zero in the A register, the function was rejected; everything has been released already, and we return to IMFILE to start over. Otherwise, call READY again to wait until the printer is ready and not busy.

The code up to this point could not be included in the overlay, because of the calls to LPC and READY. But now we call overlay B.FILL to build the line image, a purely internal process with no delays.

224g. B.FILL

At FILLX, we alter the first word of table ALPH so that display code 00B will be translated and sent to the printer as BCD 16B rather than 60B (blank). In actual printing (and BCD punching) of display code, 00B has to be treated as blank, as well as display code 55B; so this leaves one of the 64 possible BCD codes, namely 16B, unrepresented and impossible to output. The percent sign is never printed, but it may be on the printer train, and we want to have a distinct code, 16B, associated with it in the printer's image buffer so that at least no other code will ever cause it to print.

Now we check the disposition code to see if the printer is using a 48-, 64-, or 288-character train; according to this we branch to FILLZ, FILLC, or FILLR respectively.

The line image is to be set up in IM288 through IM288+287, and subroutine PRT will be altered temporarily to transmit from there rather than from IMAGE as usual. This is because an ordinary print line can never be more than 155 bytes long, and 1IR will assemble correctly as long as there are 160 bytes above IMAGE in the PP memory. But here we deal with a 288-byte image, that might overflow the memory.

So the 288-character is simplest. The 288-byte train image is already at IM288 (but the table is empty in the current version of 1IR-1IS) and we branch straight to FILLR. For the 48- and 64-character cases, we use the tables at IMAGE48 and IMAGE63 to build up a 144-byte image beginning at IMAGE. The 48-character case is trivial, as we just copy the 24 bytes of the table six times over, corresponding to the sixfold repetition of the character set on the train. The 64-character case is more complicated, and not worth describing in detail, but we build a 144-byte image corresponding to the arrangement of the train as given in the first descriptions of the 512 printer. For both 48 and 64 characters, we come to FILLK after the 144-byte image is formed, to expand it to a 288-byte image beginning at IM288. The two characters in each byte are separated and made the right halves of two consecutive bytes, of which the left halves contain zero.

For all three types of train, we arrive at FILLR; store 288 in WC, as the length of the "line image" to be sent out; alter an address in subroutine PRT so that it will find the line at IM288 ff. rather than IMAGE ff.; and return to the main part of the subroutine at FILLQ. Here we call PRT to transmit the image, then restore the first byte of table ALPH to its original value, 6055B, and restore the address IMAGE in subroutine PRT.

At IMFILA, we call READY to wait until the printer is ready and not busy. On return, send either 13B or 14B to the printer as a function code, according as the disposition code is for 288-character printing or not. These two functions select and deselect, respectively, extended array printing. If the return from FCN is with non-zero in the A register, the function has been rejected and everything released. Then we must call RES again to reserve and connect everything, and return to IMFILA to try again. But if the return from RES is with non-zero in the A register, we must call subroutine LPC to give other files their turns, and then go to IMFILB to try RES again.

When the function code has been accepted, we get to IMFILF and call subroutine READY once more. On return, send function 30B to the printer to cancel 8-line spacing and auto-page-skipping. If the return from FCN is with non-zero in the A-register, go back to IMFILB and start over as if the 13B or 14B function had been rejected. If the 30B function is accepted, continue by calling READY again, and then checking FL.AUTO. If =0, go to IMFILC; otherwise, we must renew the auto-page-skip select by sending function code 5 through subroutine FCN. Once again, if this is rejected, return to IMFILB and start all the functions over; if accepted, continue to IMFILC, after calling subroutine READY.

At IMFILC, check FL.8LL; if =0, go to IMFILD; otherwise, we must renew the 8-line spacing selection by sending function code 10B through subroutine FCN. If this is rejected, return to IMFILB and start all the functions over; if accepted, continue to IMFILD.

At IMFILD, we release the channel, 6681, and printer through subroutine REL, and then exit from IMFIL to the return address that was stored on entry at DEEP1.

Entry Information

Only the disposition code, in F5. The printer, 6681, and channel, are disconnected, unselected, and unreserved.

Exit Information

None. The hardware is again disconnected, unselected, and unreserved.

Non-Hardware Overlay Subroutines

B.MSG

225. This is called with one or two message numbers in D.T0, to get the corresponding messages written on the system dayfile or displayed on the J- and B-displays. D.T0 contains 6/z, 6/y where y is the number of the first message and z is the number of the second or 0 if there is only one message.

226. First, at MSGZ, the 9th through 14th words of the FET are read to IMAGE through IMAGE+29. This puts the equipment type and EST ordinal, in display code, in IMAGE and IMAGE+1, two blanks in IMAGE+2, the file name in IMAGE+3 through IMAGE+6, and either zeroes or a previous B-display message in IMAGE+7 through IMAGE+29. Now D.T0 is picked up, 6/0,6/z formed and saved in MSGY+1, and y considered. It is logically 1/ya,5/yb where yb is the message number and ya is 0 for a B-display message or 1 for a system dayfile message. After yb is isolated, it is stored in D.T1, and used to consult table MSDIC, fetching the starting address of the message and storing it in D.Z1. Then, if ya is 0, the message is copied into IMAGE+7 ff.; if it is 1, into IMAGE+2 ff.; in either case, the process stops at the first zero byte in the message. Then, if ya is 0, control goes to MSGB; otherwise subroutine R.DFM is called to add the message beginning at IMAGE to the system dayfile. This gives the device type and number, and one of messages CFRB, CMEA, PRES, and CFRA.

227. If ya is 0 and control goes to MSGB, all the bytes after the end of the message, up to and including IMAGE+29 are zeroed. Then IMAGE through IMAGE+29 are written back to the 9th through 14th words of the FET. This means that one of the messages XFRB, STSA, CMEB, REJ, XMS, PRER, and XTRAY has been copied into the right three bytes of the 10th word, and all of the 11th through 14th words of the FET. This extends the J-display for the equipment, and causes a B-display if one of the four lines for the control point is free. The message will be cleared away the next time subroutine CLERMES is called.

228. After disposal of the message one way or the other, MSGY is reached and 6/0, 6/z picked up. If z is 0, there was only one message and there is an exit from the subroutine. Otherwise, 6/0,6z is stored in D.T0, and B.MSG returns to MSGZ to deal with the second message just as if D.T0 contained 6/z,6/y where z was 0, and it were dealing with the first message. The next time it gets to MSGY, MSGY+1 will contain 0 and there will be an exit from the subroutine.

Entry Information

6/z,6/y in D.T0 where y is the first message number and z is the second message number or 0 if only one message. In both z and y, the right 5 bits are the message number proper, and the left bit is 0 for a B-display message or 1 for a dayfile message.

Exit Information

None.

B.MOVBF

229. This is called after subroutine MAYMOV if MAYMOV shows that a buffer move is possible and desirable. But the first thing done in B.MOVBF is to call MAYMOV again. This is because in one of the two places where we consider buffer moving, between STARTK and START7, MAYMOV must be called and if the answer is favorable, FILWT is called to wait for file activity to cease before calling B.MOVBF. Buffer assignments may have changed during the wait, so MAYMOV must be called again, but the chance of getting a good answer again is good, so it is worth loading B.MOVBF: hence, the second MAYMOV call is left in the overlay rather than in the permanent part of 1IR, and here it is.

230. The A register will be non-zero on exit from MAYMOV if a buffer move is not to be done, and in this case, there is simply an exit from B.MOVBF. If the A register is zero, MAYMOV will have left the number of our buffer in D.T5, and the number of the buffer we are to move to in D.T6; and the interlock will exist for the buffer assignment registers, which will have been copied into KEY through BUFAS+24. Now the distance by whichthe contents of our present buffer are to be moved down in central memory is calculated and they are moved down, 8 words at a time. Then the buffer assignments are changed in BUFAS ff. Then subroutine PUT is called to release the interlock and replace the buffer assignment registers in CP+77B through CP+104B. Finally the distance by which the contents of our former buffer were moved down is subtracted from the FET pointers, and the new FIRST address is copied to subroutine FIRST.

Entry and Exit Information

None.

231. B.DROPBF

This is called by overlays B.FINIS, B.RTER and B.TER to drop the buffer that was being used for the current file, and leave it available. First subroutine FETCH is called to get access to the buffer assignment registers and copy them to KEY through KEY+4 and BUFAS through BUFAS+24. 1 is subtracted from KEY+3, the count of buffers in use. Knowing the FIRST pointer of the current FET, and having the starting address of buffer number 0 in ZERBUF, calculate the number of the current buffer, n, and then zero BUFAS+n to show that buffer number n is now free. Then call subroutine PUT to replace the buffer assignment registers in control point +77B through 104B, and release the interlock on them. Then exit.

Entry and Exit Information

None.

231a. B.SETP

This is called to set the buffer pointers, both in the FET and in the PP, to indicate either an empty buffer, or a buffer in which the first n CM words contain information; IN contains n, which is 0 for an empty buffer, on entry. Thus, we set OUT = FIRST and IN = FIRST+n.

Entry Information

IN contains the number of CM words of information which the buffer is, according to the IN and OUT pointers, to appear to contain.

Exit Information

OUT in the PP contains 0.

232. B.WUMES

This is called to add a message of three CM words to the pre-OUTPUT file for the job being read in on the current card reader file. D.T0, on entry, contains the starting address of the message. The number of words of message that have already been put in the file is contained in bits 48-58 of the 7th word of the FET. 3 is added to this, but if it becomes greater than 126 (the mini-buffer has room for only 127 words) it is restored and exit taken. This means that messages after the 42nd on the pre-OUTPUT file are lost. If there is room for the new message, the mini-buffer is located. If there are n card readers, the FET number in PS will be between 1 and n; call it k. Then the mini-buffer associated with the current FET is 200B words long and begins at p-200B*k where p is the starting address of buffer number 0, contained in ZERBUF. The first of the 3 words into which the new message must be written is at p-200B*k plus the number in bits 48-58 of the 7th word of the FET, minus 3, as 3 has already been added to that number for this message. The message is written to central memory and followed by an exit.

Entry Information

The starting address of the message in PP memory, in D.T0.

Exit Information

None.

233. B.AMSG

This is called by overlays B.TER and B.RTER to write on the system dayfile, after a file has been completely read, punched, or printed, an accounting message giving the job name and the number of cards read or punched, or of lines printed. First D.Z2 is set to point to the beginning of one of the messages DAX, DAY, or DAZ, according to the type of file indicated by TYPE. Then the job name is gotten from CURFET through CURFET+3 and its 7 characters, preceded by a blank, copied into the first four bytes of the message area. A normal job name contains exactly 7 characters, but there are names like DIS created by the system, which could produce zero in the third or fourth bytes of the message; so a check is made for this using 5555B instead of 0000B if necessary. The message area contains 5 display code zeroes, into which decimal digits must be added to give a decimal count of cards or lines read, punched, or printed. So subroutine FIG is called four times, with divisors 10000, 1000, 100, and 10 in the A register. FIG subtracts the divisor, as many times as possible without getting a negative result, from the line/card

count in bits 0-23 of the 7th word of the FET (CT and CT+1). On exit from FIG, the quotient, i.e. the number be added to the corresponding zero in the message, is in the A register, and is added to the appropriate 6-bit area in the message. At the end, the units addend is left in CT+1 and is added directly to the last zero in the message without going through FIG. Then subroutine R.DFM is called to write the message on the system dayfile, and an exit is taken.

Entry and Exit Information

None.

234. B.FCIA

This is really just two tables, one a table of visible character images, and the other a table of Hollerith column images, for all the letters and digits. The second table (HOLLER) could just as well have been provided as part of overlay B.INIT; the first one (FCIA) is used by both B.INIT and B.LINIT, hence the advantage of providing it as part of a separate overlay. In execution, this overlay subroutine immediately exits from itself, so that executing it simply copies the tables into the high end of the overlay area.

Entry and Exit Information

None.

235. B.REW

This is called from overlays B.MTER and B.FINIS to rewind a file. The FNT entry, whose address is in DEST3 is copied into MYFNT through MYFNT+14. Then the status byte of the FNT entry is set to 53B, for rewound, and subroutine WTFET is called in to set the status of the FET likewise. Then the level number field of the FNT is set to 0. Then the current RBT pointer in the FNT entry is set the same as the initial RBT pointer. Then, with the use of that pointer and the size of central memory (which is given, divided by 100B, in bits 0-11 of the CM word at P.RBR) the first word of the first RBT word pair is read; the initial RBT byte number is taken from bits 36-38 of this word, and this is made the current byte number in the FNT entry, and the current RBT ordinal in the FNT entry zeroed. Then the PRU number in the FNT entry is set to zero, the entry is copied back to central memory, and an exit taken.

Entry and Exit Information

None.

236. B.SKFNT

This is called at three points to find a free FNT slot, if there is one, and claim it for JANUS:

1. In overlay B.FORAGE, before starting to look for a previously idle card reader with a job card ready to be read from it; if such a reader is found, an FNT entry will be needed for the new job file.

2. In the main loop just after CRD, when it is found that there is a card read file, but the FNT entry address (in DEST3) is 0; this must be a file that follows on the heels of its predecessor without a break; everything in our setup was kept intact when the predecessor was terminated except that its FNT entry became that of an input file at control point 0, and now a new entry is needed.

3. In overlay B.RTER, in terminating a card read file, if a pre-OUTPUT file has to be written an FNT entry is needed for it.

237. The control point is inserted into the model at MESS of the first word of an FNT entry for a local file called MESSAGE. Then the FNT pseudo-channel is secured and the FNT scanned for an empty slot. If none is found, the FNT pseudo-channel is dropped and exit taken with non-zero in the A register. If one is found, its address is saved at PS1, and an FNT entry copied into it, composed of the first-word model in MESS through MESS+4; 60/0 as the second word; and 60/1 as the third word. Then the FNT pseudo-channel is dropped and an exit taken with 0 in the A register.

238. It is possible that if there is more than one card reader in the system, there could be more than one FNT entry for a file called MESSAGE at one time at the JANUS control point. But every FNT entry is identified, as far as JANUS is concerned, by its address and not by the name in its first word; and JANUS never releases an FNT entry to control point 0 without giving it its proper name.

Entry Information

None.

Exit Information

Non-zero in the A register if there was no room in the FNT for a new entry; otherwise zero in the A register, a dummy entry in thr first vacant FNT slot found, and the address of that entry in PS1.

238a. B.SETFT

This is called by overlay B.FNT, for a print or punch file, and overlay B.FORAG, for a card reader file, to secure a buffer for the file and set up the pointer words of its FET. D.LIMIT contains the amount to be subtracted from BUFLG to give the actual length of the buffer; this subtrahend is the length of the print line or card image area to be left after the end of the buffer. PS contains the FET number, and FETAD contains 16 times this, i.e. the address of the beginning of the FET.

We call subroutine FETCH to secure access to the buffer assignment words in the control point area, and to fetch them. Then read the control point word in which 1IR may earlier have requested a new buffer from 1IQ, the field length then being exhausted. If the first byte of this word contains 2, a buffer has been assigned; we can now use it; and its number is in the last byte of the word, i.e. D.Z7. So we zero the first byte of the word to show acceptance, and branch directly to SETFETB.

Otherwise, we must look for a free buffer. From SETFETA, if there is at least one buffer free, we branch to SETFETC; otherwise, we set the first byte of the word at control point +73B to 1, to request a new buffer from 1IQ; then call subroutine PUT to release the buffer assignment registers, and exit from B. SETFT with a negative number in the A register to indicate failure. At SETFETC, we increase by one the count of buffers now in use, and scan the assignment registers to find the lowest free one. This brings us to SETFETB with the free buffer number in D.Z7.

At SETFETB, we claim the buffer by copying the FET number into the corresponding register, and call subroutine PUT to copy the buffer assignment registers back to the control point and release our interlock on them. Then multiply the buffer number by the length of a buffer, add the address of buffer number 0, and put the result in subroutine FIRST (this is the relative address of the start of the buffer) and also in a FIRST pointer for the FET, with the error processing bit set to 1 (if this EP bit were 0, a parity error in a JANUS disk file would cause JANUS to drop). Then copy the FIRST pointer word to the FET; zero the EP bit, and copy the same word to the FET as IN and OUT pointers. Finally, add BUFLG minus the content of D.LIMIT (as explained above) to the pointer to get the value of LIMIT, and copy the LIMIT pointer into the FET; then exit with a positive number in the A register to indicate success.

Entry Information

D.LIMIT contains the difference between BUFLG and the wanted buffer length; PS contains the FET number; FETAD contains the relative address of the start of the FET.

Exit Information

The A register is positive for success, or negative for failure.

Hardware Subroutines

FCN

239. This is called with a format function in the A register, to be sent to the punch or printer for the current file. The equipment has already been checked for readiness; however, the code is saved and subroutine STATUS called to re-check it for busyness. If it is busy, STATUS is called repeatedly until it is not busy. It seems as though this could never happen. On entering FCN, the equipment has already been connected, and STATUS does not release it or its channel.

240. At FCN1, the function code that was in the A register in the first place is picked up and subroutine CFR called to send it to the equipment. If the return is with 0 in the A register, the function has been accepted; the equipment and its channel have not been released, and we exit from FCN with 0 in the A register.

241. If the exit from CFR is with non-zero in the A register, the function has been rejected and the equipment and its channel have been released. The exit from FCN is with 1 in the A register.

Entry Information

The A register contains the function to be sent to the equipment, and the channel, 6681, and equipment have been reserved, selected, and connected.

Exit Information

If the A register contains 0, the function was accepted and everything is still reserved and connected; otherwise the A register contains 1, and the function was rejected, and everything has been released.

RES

242. This subroutine reserves the channel, selects the 6681, and connects the equipment for the current file. First it requests a channel. DEST1 contains one or two channel numbers for the equipment; this byte is copied to D.T1, to form part of the request. D.T2 may also contain one or two channel numbers in a channel request, so this cell is zeroed to prevent it. Then R.RCH is called. On return, the number of the channel that was granted is in D.T1, and is at CHAN. All the instructions in 1IR that involve channel numbers are listed in table RESA, beginning at RESA+1. (They all contain the symbol NULLCH for nominal 0 channel number, to produce a list of the locations of these instructions in the cross-reference table.) Initially they all refer to channel 0, and their channel references are altered only by subroutine R.STB, in such a way that they are all changed to refer to the same channel. So now CHAN can be matched against the channel number in one such instruction, RES4. If they match, then all the channel-referencing instructions in 1IR refer to the same channel, the one just reserved, and RES goes straight to RES3. Other-

wise, subroutine R.STB is called to alter all the instructions listed in table RESA to refer to the channel whose number is in CHAN. Then RES proceeds to RES3.

243. At RES3, the 6681, whose select code is 2000B, is selected and the equipment, whose connect code is in DEST4, connected. Next, if TYPE shows that a card reader is involved, function code 0001B is sent on the channel to negate automatic Hollerith-to-BCD conversion. Next subroutine ACC is called to see whether the unit has accepted the last code sent; this is the negate-conversion code for a card reader, or the connect code for a punch or printer. If the return from ACC is with 0 in the A register, the code was accepted, and the channel and equipment are still reserved and connected. Otherwise, rejected, released, and disconnected. For a card reader (according to TYPE), exit with 0 in the A register, to switch to a different unit if possible (the fact that SW is called with bit 1 of the A register a 0 means that SW may try to switch even if the last-known status of the unit was busy). Then an exit is taken from RES with 1 in the A register.

Entry Information

None, but the channel, 6681, and device are unreserved, unselected, and unconnected.

Exit Information

If the A register contains 0, they are reserved, selected, and connected; otherwise, the A register contain 1 and they are still unreserved, unselected, and unconnected.

CFR

244. This is entered with an equipment function code in the A register, which is immediately passed to the equipment by a FAN instruction. Then the return address is moved to the return address cell of subroutine ACC, and branch to ACC+1 is taken as though RJM ACC had been done in the first place rather than RJM CFR.

Entry and Exit Information

The same as for subroutine ACC, except that CFR is entered with equipment function code in the A register.

ACC

245. This is entered immediately after a function code has been sent to an equipment by a FAN or FNC instruction, to check whether the code was accepted. The 6681 status word is read from the channel, and if bits 0, 1 and 2 are all zero, there is an immediate exit with 0 in the A register. If bit 2 is 0, subroutine MAYMES is called, which will probably put the message REJECT FUNCTION on the system dayfile and the B-display; then subroutine REL is called to deselect

the 6681 and release the channel; then there is an exit from ACC with 1 in the A register. If bit 2 of the 6681 status is 1, control goes to ACC7 and subroutine MAYMES is called which will probably put the message 6681 XMSN PARITY ERROR on the system dayfile and the B-display; then after a one second wait (in the hopes that this allows any other devices that might be using the channel at the moment to finish transmitting data successfully) a clear channel instruction is given. Then at ACC6 subroutine REL is called to deselect the 6681 and release the channel, followed by an exit from ACC with 1 in the A register.

Entry Information

None, but the channel, 6681, and device are reserved, selected, and connected respectively, and a function has just been sent to the device.

Exit Information

If the A register contains 0, the function was accepted and the channel, 6681, and device are still reserved, selected, and connected. Otherwise, the A register contains 1 and the channel, 6681, and device are released, deselected, and disconnected.

STS

246. This reads the status of the device for the current file, stores it in STAT, and exits with it in the A register. Note that for punches and printers, STS is called only through subroutine STATUS.

Entry Information

None; but the channel, 6681, and device are already reserved, selected and connected.

Exit Information

The status of the device in STAT and in the A register; the channel, 6681, and devices are still reserved, selected, and connected.

STATUS

247. This subroutine is used to call STS for a punch or printer. It calls STS, and then exits with the status in the A register if bit 0 (HS.READY) is 1, and if, for a punch, bit 6 (HS.NFEED) is 0. Otherwise, before exiting with the status in the A register, it calls subroutine MAYMES, which probably puts the message NOT READY on the J and B displays.

Entry and Exit Information

As for subroutine STS.

REL

248. This is called to deselect the 6681 for the current file and release its channel. On entry, they are selected and reserved respectively. The 6681 deselect code is issued with a FAN instruction. With the current channel number, R.DCH is called to end this PP's reservation of the channel. Finally, PAUSE is called to allow for storage relocation. Whenever an equipment turns out to be not ready for some reason, REL is called immediately afterwards, so calling PAUSE from REL prevents a storage move hang-up from occurring because a device is persistently unready.

Entry and Exit Information

None, but the channel and 6681 are reserved and selected on entry, and released and deselected on exit.

RELCR

249. This is called, rather than REL, for card reader files. First, RELCR tries to send a 0000B code (clear and disconnect) to the reader by subroutine CFR. If this is accepted, the return from CFR is with 0 in the A register, and subroutine REL has not been called by CFR. So REL is called before exiting from RELCR. If the return from CFR is with non-zero in the A register, the function was not accepted by the card reader, but REL has been called from CFR. So we exit immediately from RELCR.

250. It is hard to see why this is done, instead of just using REL for card readers as is done for printers and punches. The only possible reason seems to be that if a reader were connected through a 3649 controller to two 6681's on two channels, and it were connected on one 6681 without the 0000B code being sent later to release it, the other 6681 would never be able to connect to it. However, the 0000B can be rejected, and in that case RELCR does nothing further about it. If the rejection is merely because the reader is busy, then the 0000B code has not been sent and it has not been disconnected.

Entry and Exit Information

Same as for subroutine REL.

NRD

251. This subroutine is called:

1. just above CRD7E, in the course of continuous card reading, when a reader is found to be not busy but not ready.

2. in overlay B.FORAG, when a reader that was not previously in use is found to be not ready, but not because of either fail-to-feed or tray-empty status.

Thus it is called when a card reader is not ready for an unexpected reason, and it gives the appropriate message.

252. If bit 10 of the status byte is 1, indicating a compare error on the last card read, control goes to NRDA, where (unless LSTAT already contains UNREADY) subroutine MAYMES sets LSTAT to UNREADY, puts the message COMPARE ERROR on the dayfile, and puts the message RE-READ LAST CARD on the B-displays. Then FL.IMAGE is zeroed; the card image that is now stored after the end of the buffer is that of the card that gave the compare error, which will be re-read; so this image is discarded by zeroing the flag. Then NRD goes to NRDB to drop the card reader, using subroutine RELCR, and exit from NRD.

252a. If there is no compare error bit, bit 5 of the status byte is tested for tray-empty status. If this is 1, NRD goes to NRDC. Here FL.IMAGE is checked; if it is 1, we just go to NRDB to drop the card reader and exit from NRD. FL.IMAGE indicates that card image is still in hand; it will simply be used now, and the next time a card is read NRD will be called, FL.IMAGE will be found = 0, and a message given.

253. But if there is no compare error, the tray is empty, and FL.IMAGE is 0, there are no more cards and it must be taken seriously. Now it is certain that control has come to NRD from just above CRD7E, not from overlay B.FORAG, as NRD is not called from the overlay if the tray-empty status bit is 1. If the FET status in CURFET4 is not INITA (=3) the tray has become empty in the middle of a file, and this deserves a message; so subroutine MAYMES is called, which will, unless LSTAT already contains UNREADY, set LSTAT to UNREADY and put the message TRAY EMPTY on the B-display. Then NRD goes to NRDB to drop the card reader and exit from NRD.

254. But if the FET status is INITA, the situation on the card reader must be this:

1. An end-of-file card terminating a job file has previously been read in this card reader, and FL.IMAGE was found to be 1. So the reader was not dropped, but the FET status was set to INITA, expecting that the card already in the hold would be a job card.
2. The card in the hold, however, turned out to be a binary card, probably an extra 6-7-8-9 card; so it was ignored and the next card awaited. This could have been repeated any number of times.
3. However, the series of binary cards is now ended and the tray is empty. What should be done, then, is to make the situation what it would have been if the tray had become empty immediately after the 6-7-8-9 card that closed the preceding file; that is, if FL.IMAGE had been 0 when we logically read the 6-7-8-9 card. So a branch is taken to NRDD; RELCR is called to drop the card reader logically, and then overlay B.RTER. In that overlay, because FL.IMAGE is 0 and CURFET4 contains INITA, everything associated with this FET will be dropped, without creating a new job file in the input stack.

<u>Entry Information</u>

None, but the channel, 6681, and reader must be reserved, selected, and connected on entry.

<u>Exit Information</u>

The A register contains 0, a fact used just above CRD7E. The channel, 6681, and reader are all released.

RCD

255. This subroutine is called in overlay B.FORAG to read the first card through a reader that was previously empty and not in use, and at CRD7E for all other cards. It is entered with zero or non-zero in the A register; if non-zero the next card physically will be read and saved in CM after the buffer. If FL.IMAGE is 1, it will be reset to 0 and the card image that was previously in the hold following the buffer copied to IMAGE through IMAGE+79 in PP memory.

256. On entry to RCD, the A register is stored at D.Z1. Then if FL.IMAGE is 0, a branch to RCDA is taken and HAVECARD is zeroed to show that a card will not have been read logically when the return from RCD is taken. (A new card will, perhaps, have been read physically, but it is still unclear whether it gave a compare error or not; that cannot be ascertained until the card reader next becomes not busy, normally when the next card is just about to be read physically). If FL.IMAGE is 1, the image of the last card physically read, which is in CM following the buffer,is copied into IMAGE through IMAGE+79. Then FL.IMAGE is set to 0 and HAVECARD to non-zero, to show that a card has been logically read.

257. After RCDA, if D.Z1 contains 0, the card reader was not ready when RCD was called, so a return is made from RCD. Otherwise, it was ready, and the next card is read from the card reader to IMAGE+80 through IMAGE+159. Then subroutine RELCR is called to release the card reader; then subroutine SALARM with AL.CR (=33) in the A register, is called in to set the alarm for this file 33 milliseconds ahead of the current value of the millisecond clock. Now, if this card produced a "compare error" status on the reader, this status probably will not appear until the reader next becomes not busy; at that time the status will be "not busy but not ready", and the card image will be rejected by the calling of subroutine NRD, just below CRD5X. However, it also seems to happen occasionally that the reader status is "ready and not busy" even though it is also "compare error", indicating that the card image about to be read into the PP may be faulty. So to guard against the possibility that we have just read such a card, we check STAT, which contains the reader status as it was just before reading the new image in the PP. If STAT shows compare error status, exit from RCD and discard the image just read into IMAGE+80 through IMAGE+159. Otherwise, move it to the 16 CM words immediately after the buffer, whence it will be recovered when it is certain that it did not give a compare error. Then set FL.IMAGE to 1, indicating the waiting card image, and exit from CRD.

Entry Information

If the A register contains 0, the card reader is connected and ready, and the channel and 6681 are reserved and selected. Otherwise the reader is not ready nor connected, and the channel and 6681 are not reserved and selected. FL.IMAGE is 1 if a card is to be logically read, and 0 if not.

Exit Information

HAVECARD contains non-zero if a card has been logically read, and zero if not. FL.IMAGE is 1 if a new card image has been read and saved in central memory, and 0 if not. The channel, 6681, and reader are not reserved, selected, or connected.

PRT

258. This subroutine is called from the main loop to print a line, or from subroutine PCHOLD/PCHNEW to punch a card in BCD mode. The line or card image is in IMAGE ff. in display code (or in extended array printer code; see below) and the number of bytes in the image is in WC. Though SCOPE at the moment provides for 512 printers only with 64-character trains, 48- and 288-character trains will eventually be provided for, and with them it becomes possible for a character to be unprintable. If it is sent to the printer, it causes an error status that could be confused with hardware trouble. So PRT has been coded to check the print line image on the following assumptions:

1. The disposition code for a file being printed with a 48-character train will be 43B, and with a 288-character train 44B. These codes in fact never occur at present.
2. For a 48-character train, the allowed characters correspond to display codes 01B through 54B and 56B through 61B. (Blank, for display code 00B or 55B, is not included in the count of 48.) Thus any display code over 61B is unprintable.
3. For a 288-character train, the allowed characters are 0000B through 0441B, of which 0040B is a blank by hardware necessity, and 0000B is also to be treated as blank, by SCOPE convention. Thus any byte over 0441B (=289) is unprintable.
4. If the disposition code is not 43B or 44B, all 64 possible display code characters are printable, on the 501 or 512 printer, or are punchable in BCD mode.

On entry to PRT, then, we check the disposition code and set the program switch at PRTAX accordingly. Then connect the unit, activate the channel, and zero the byte counter (D.Z1) and error counter (D.Z6).

At PRTA, pick up the next byte of the image.

1. for a 288-character printer, go to PRTAZ. If the byte is zero, change it to 0040B (blank) and go to PRT3. If it is unprintable, do the same and add one to the error counter.
2. for a 48-character printer, check both halves of the byte, and if either half is unprintable replace it by 00B, which counts as blank, and add one to the error counter. Then go to PRTAW.
3. for a 64-character printer or punch, go straight to PRTAW. At PRTAW, convert according to the left half of table ALPH. Then go to PRT3.

At PRT3, output the byte, whether one character (288-character printer) or two (all other cases) and wait till the channel is empty. Step the byte counter, compare it with WC, and return to PRTA if necessary. When the line is completed, disconnect the channel and exit from PRT. But if the error count is non-zero, set FL.CMESA=1 before exiting. This will cause the print line to be followed by an extra line with an X below each unprintable position in the original line, and then the message "UNPRINTABLE CHARACTERS IN PRECEDING LINE".

Entry Information

The image is set up in IMAGE ff., and the byte count is in WC. The channel has been reserved, the 6681 has been selected, the punch or printer has been connected, and a BCD mode control function has been sent to the equipment function if it is a punch.

Exit Information

None, but the channel and 6681 have not been released.

PCHOLD/PCHNEW

259. This is practically one subroutine, which punches a card whose image is in IMAGE ff. The reason for having two entry points is that it may be called to punch the card image most recently set up (PCHNEW) or the image of the preceding card (PCHOLD) when re-punching due to a compare error is called for. The subroutine must know whether the card is BCD or binary; FL.BCDN is set 1 if the most recent card is BCD; while FL.BCDO is 1 if the preceding card is BCD. So whichever way the subroutine is entered, at PCHA the relevant flag bit is in the sign position of the A register, and a branch is taken to PCHB if it is 1. At PCHB, WC is set to 40, as the image of a BCD card is only 40 bytes long; then function 2 is sent through subroutine FCN to select BCD punching. If the return from FCN is with non-zero in the A register, there is an immediate exit from PCHOLD/PCHNEW with non-zero in the A register indicating non-success; the channel, 6681, and punch will already have been released. Otherwise, subroutine PRT is called to punch the BCD card, and go to PCHC.

259a. If the relevant flag for BCD is 0, there is no branch from PCDA to PCDB, but 1 is sent through subroutine FCN to select binary punchung. The non-zero exit possibility from FCN is the same as described in the preceding paragraph. Then the 80 bytes beginning at IMAGE, are punched and control goes to PCHC.

260. At PCHC, subroutine SALARM is called to set the alarm for this file AL.CP (=200) milliseconds ahead of the current value of the millisecond clock, and then there is an exit from PCHOLD/PCHNEW.

Entry Information

The card image is in IMAGE ff., and FL.BCDN or FL.BCDO, depending on which entry point is used, is 1 if it is to be punched in BCD and 0 otherwise. The channel, 6681, and punchare already reserved, selected, and connected.

Exit Information

If the A register contains 0, the card has been punched and everything is still reserved, selected, and connected. Otherwise, it has not been punched because the unit rejected a 1 or 2 function, and everything has been released and disconnected.

RUDDY

261. This subroutine is a variant of READY, which waits, with file-switching as necessary, until the current file's 512 printer is ready and not memory-busy. This condition, rather than ready and not busy, is what has to be awaited before sending a line to the 512 for printing.

First we save the return address at DEEP (which is necessary because LPC is going to be called, so that RUDDY might serve another file before completing action on this one); then call subroutine STATUS and check the returned status for ready and memory not busy. If the status is good, exit immediately from RUDDY, by going to READYX, calling CLERMES, and returning to the address stored at DEEP. Otherwise, call subroutine REL to release the printer, 6681, and channel, and call subroutine SW with bit 6 of the A register (the one that corresponds to the memory-busy bit in the printer status byte)=1. SW will try to switch the file to another 512 printer if this has been requested by a type-in, but not if that bit of the status byte is still 1.

On return from SW, we call subroutine LPC, to give other files their turns, and subroutine RES, to return the hardware to the reserved and selected condition it was in when we entered RUDDY. If the return from RES is with 0 in the A register, all is well and we return to RUDDYB to repeat the cycle. If with non-zero, the A register must in fact contain 1 (see RES) and everything is unreserved and unselected. So we return to RUDDYC and call SW, this time with 1 in the A register; SW will now try to switch (if requested by a type-in) even if the printer status is memory busy. If subroutine RES is failing, we cannot even read the printer status byte, so waiting for it to change would be hopeless.

Entry Information

None, but the printer, 6681, and channel are connected, selected, and reserved.

Exit Information

The same as entry information, but the printer is known also to be in ready and memory not busy status.

16.1.3 1IS and 2IS - General

1IS is a PP program, always loaded at 1000B. It is called twice by 1IQ when it is initializing the JANUS control point; once to communicate the length of the overlays it (1IS) will later put into the field length, so that 1IQ will know how much FL to ask for; and a second time, after the field length is obtained, to store the overlays and their directory in the field length. 2IS is merely an overlay to 1IS, loaded at 2000B, to contain additional overlays to be copied to the field length.

Function

1IS copies out of itself, into an area immediately following the FET's in JANUS field length, a number of overlays for later use by 1IR. It also stores in RA+10B through RA+17B an index to these overlays. It is called by 1IQ to communicate the length of these overlays, so that 1IQ will know how much storage to request in the first place.

The overlays, since they will work with 1IR, contain frequent references to address symbols in 1IR. So the most practical way to assemble 1IS and 2IS is as a second and third segment of 1IR, with ORG 1000B and ORG 2000B at their beginnings. For the way 1IR uses an overlay, see subroutine EXECUTE in 1IR.

Entry Information

The input register contains

```
VFD      18/0H1IS,3/0,3/c,24/0,12/v
```

where c is the control point number, and v is 0 the first time 1IQ calls 1IS, and, the second time, the address of the first cell in central memory after the last FET.

Exit Information

When 1IS has finished and is about to drop itself, it copies its input register into CP+76B. The only significance of this is to make bits 48-59 of that word non-zero, so as to inform 1IQ that 1IS is finished.

Bits 0-11 of the word at RA+17B will then contain the total length of the overlays provided by 1IS, in CM words. If bits 0-11 of the 1IS input register were 0, this will be the only information furnished by 1IS, although it will also have written into the rest of RA+10B through RA+17B.

If bits 0-11 of the 1IS input register were not zero, the overlays will have been stored beginning at RA+v, where v is the content of that byte. RA+10B through RA+17B will then (except for the rightmost byte of RA+17B, as noted

above) contain an index to the overlays. Overlays are numbered 1 to 39, but need not all exist. The address of the beginning of overlay n is found by dividing (n-1) by 5, producing quotient q and remainder r. In the word at RA+10B+q, bits 48-12r to 59-12r contain either 0, in which case the overlay does not exist, or the address of the overlay relative to RA.

Other Programs Called

2IS, as an overlay containing further overlays to copy into the field length, but no code for 1IS to execute.

Narrative

The action of 1IS begins at IIS, which is found in the assembly listing of 1IR and 1IS at the end of the sequence of pages that do not have L in the left margin, before the pages of overlays that do have the L. 1IS reads its input register, sets the control point address, and checks the control point error flag, dropping the PP if this is non-zero, at DRAP.

At IISA, we zero D.FNT so that it will certainly not show a drop in address the first time it is used below. At IISB look successively at bytes INDEX through INDEX+38. If a byte is zero, we assume we have reached the end of the index, and go to IISDX. Otherwise, if the byte is equal to or greater than the preceding byte, which has been stored in D.FNT, go to IISE. But if the new address is less than the preceding one, we must have reached the break between 1IS and 2IS. So we call in overlay 2IS and return to IISA, to zero D.FNT and then continue normally.

At IISE, we process a byte from the index by storing it in D.FNT, for comparison with the next byte, and as a pointer to the start of the overlay; also insert it in the CWM instruction at IISC to control the copying into the field length. Next pick up the first byte of the overlay, which gives its length-1 in CM words (minus 1 because this byte is mostly used by 1IR subroutine EXECUTE, where a value that excludes the control word at the beginning of the overlay is more convenient). Put the total length in D.Z4, to control the CWM instruction at IISC, and add it to the cumulative total of overlay lengths in INDEX+39.

Now if bits 0-11 of the input register contain 0, this is all that has to be done about this overlay, and we branch to IISD, to step ahead in the scan of INDEX through INDEX+38. Otherwise, this byte contains the CM address of the beginning of the area to which the overlay is to be copied. Initially this is the address of the first word after the last FET, and after copying each overlay to CM its length is added to this byte. So now store this address back in the cell of the index from which the PP address of the beginning of the overlay was just taken; the PP address has already been saved at IISC+1. Finally, at IISC, write the overlay to central memory; then subtract RA from the A register, which the hardware has updated automatically, and store the result back in bits 0-11 of the input register.

At IISD, continue the scan of cells INDEX through INDEX+38 and either go back to IISB, or, having completed the scan, copy the index to RA+10B through RA+17B. Then the input register is copied to control point +76B, to tell 1IQ that 1IS has finished its work, and control goes to DRAP to drop the PP.

16.1.4 1IU General

1IU is a PP program, always loaded at 1000B. It is called by 1IR when a print file has to be backspaced.

Entry Information

The input register initially contains

```
VFD     18/0H1IU,3/0,3/n,12/0,6/p,6/c,12/f
```

where n is the control point number, p is the number of pages to be backspaced (if p = 0, 1 is used), c is the character that defines the beginning of a page, in display code (if c = 0, 34B, the format character 1, is used) and f is the FET number of the file to be affected, taken from cell PS in 1IR.

Exit Information

When backspacing is completed, or the file has been rewound without satisfying the page count, 1IU sets bits 0-5 of the first word of the FET to 17B, i.e., a write status. This false status is taken by 1IR as a signal that 1IU is finished, and 1IR can proceed with the file. Write status is never found otherwise in a print file FET, and in any case 1IR is, as far as this file is concerned, in a tight loop waiting for the 17B status to appear; so a write status could be used as a signal in some other part of the main loop of 1IR for print files, as long as a read status was set before calling 1IU. It is highly desirable to have 1IU set its completion signal in the first word rather than in the 7th word, which contains most of the flag bits, because while 1IR is waiting it repeatedly calls its subroutine LPC. This subroutine always writes the 6th, 7th, and 8th words of the FET from PP memory to the CM as it puts one file aside, but reads back the first word as well as these three when it picks a file up. So if 1IU tried to alter the 7th word, it would risk a clash with 1IR; but no clash is possible on the first word if 1IR is merely calling LPC in its wait loop.

Other Programs Called

None.

Narrative

1IU initially sets constants 1 and 3, finds the control point address, the RA and FL, and drops itself if there is an error flag. Using the FET number in bits 0-12 of the input register, it reads the FNT entry address for the file from bits 12-23 of the 8th word of the FET, and stores it in D.FNT, where it will remain throughout. Then it finds the number of words currently in the buffer, and stores it in D.FNT+2. Then it sets the FET pointers to indicate an empty buffer, with IN = OUT = FIRST, and LIMIT =

FIRST+65. The buffer is temporarily shrunk in this way so that every read on the file will read exactly one PRU; this shrinking is necessary because there is no special read-one-PRU function available for an allocatable device. The first task is to manipulate the file so that only the PRU containing the word to which OUT pointed when 1IU was called is the buffer, and to make OUT point again to that word. In the case that the buffer was empty when 1IU was called, the buffer will be made to contain the PRU last read before 1IU was called, and both IN and OUT will be made to point to the last+1 word of that PRU. So a backspace is performed over consecutive PRU's backwards, beginning with the last one read before 1IU was called, accumulating a total of the words in the PRU's backspaced over, until this total is greater than the number of words the buffer contained when 1IU was called. After backspacing over each PRU, 1IU has had to re-read it immediately to find out how long it was. So when the total of words backspaced is greater than the number to be backspaced, the desired PRU is already in the buffer, and IN points to its last+1 word. Then OUT is set to FIRST plus the difference between the number of words backspaced and the number which were to be backspaced, and 1IU is ready to begin the real job of backspacing.

This initial backspacing is done from BA to a point a few instructions below BB. Subroutine BACK backspaces the file one PRU, while subroutine READ reads one PRU into the 65-word buffer, and puts its length into D.FNT+3. First we backspace and reread the last PRU that was read before 1IU was called, and subtract its length from D.FNT+2. If this makes D.FNT+2 negative, we branch to BB. Otherwise,we backspace over that PRU again, and return to BA to backspace and reread the next previous PRU, and so on. At BB, we invert the sign of the difference, giving the number of CM words, at the beginning of the PRU just read,that must not be backspaced initially; this is added to FIRST, resulting in the proper setting of OUT.

Now OUT points to the first unprinted word in the file, and backspacing over words that have already been printed can begin. From bits 18-23 of the input register, the number of pages to be backspaced is obtained, or 1 if this number is 0, and it is stored in D.EST. From bits 12-17 of the input register, is obtained the page format character to be used, or 34B (display code 1) if this field contains 0. Then 1IU initializes the count of lines per page to 66, in D.EST+2, and zeroes D.FNT+4 and D.FNT+5. D.FNT+5 will contain the first byte of each CM word read, when the word next preceding it is checked. D.FNT+4 will be set to non-zero by subroutine READ whenever it reads a PRU giving an end of record/file response, or to zero otherwise.

At AC a check is made for OUT = FIRST. If not, scanning the current PRU backwards is not complete and we go to ACA. If they are equal, we must back up another PRU, so call subroutine BACK twice to backspace over the PRU just scanned and its predecessor, and subroutine READ to read in the predecessor. Then set OUT = IN (not = FIRST, since it is to scan backwards not forwards) and return to AC.

At ACA, OUT points to the word last looked at, so we read in the CM word preceding that one. (Though the first clause of the preceding sentence is sometimes logically but not literally true, the second clause is literally true). If D.FNT+4 does not contain zero the last word of a logical record has just been read into PP memory; in this case D.FNT+4 is reset to zero as well as the fifth byte of the word just read; if it was not formatted as a line terminator, it should have been. Next a check is made of the fifth byte of

the word just read. If it is not zero, it is not the last CM word of a line, and we go to ACB, where we save the first byte of the CM word in D.FNT+5, reduce OUT by 1, and return to continue the backward scan of the buffer. But if this is the last CM word of a line, the first byte of the next line must be the first byte of the next following CM word, which has been saved in D.FNT+5 (OUT points to this word.) So we compare the first character of the next line with the format character in D.EST+1. If they are the same, we have backspaced over a page, and go to ACE. If not the same, we have at any rate backspaced over a line, and go to ACC, where the lines-per-page count in D.EST+2 is reduced, and we go to ADB if it is not exhausted. If it is exhausted, we go to ACE as if the next following line had begun with the page format character. This may not give an exact page count if page format characters are not properly placed in the portion of the file being backspaced over, but it will prevent us from going right back to the beginning of the file in a vain search for page format characters.

We come to ACE on deciding, either by format character or by line count, that we have just finished backspacing a page. Subtract 1 from the page count in D.EST, and go to DONE if it is exhausted. If not, reset the lines-per-page count in D.EST+2 to 67, and pass to ACC where it is immediately reduced to 66; then go to ACB to continue the backward scan.

DONE is reached when we have, just above, exhausted the page count, or have reached the beginning of the file in subroutine BACK. In the former case, OUT points to the first word of the first page to be printed after the backspace; or if OUT=IN, the first word of the next PRU is that word. In the latter case, OUT=FIRST and the buffer is empty. Now copy the OUT pointer from the PP to the FET in CM; reset the limit pointer to FIRST+BUFLG-40B, and set the status in the first word of the FET to 17B. Note that the first word of the FET must be the last that 1IU alters at this point; otherwise, 1IR might get the completion signal too soon and use a wrong value for OUT or LIMIT. Then go to DROP to drop 1IU.

Subroutines

PAUSE

This is called only by subroutine FWAIT, while waiting for the FET to show not busy. It calls subroutine R.PAUSE; then resets D.RA and D.FL, and goes to DROP if the error flag is F.ERPP. If the error flag is something else, 1IU does not drop because JANUS is supposed to be completing any pending files before dropping.

Entry and Exit Information

None.

Other Subroutines Called

R.PAUSE

Registers Destroyed

D.T0 through D.T4

FWAIT

This is called during the initialization of 1IU, to wait for the file to be not busy, and by subroutine DORQ to wait for not busy after each read or backspace initiated by 1IU. It checks the completion bit in the FNT entry. When this is 1, it branches to FWAITA; otherwise, it waits one millisecond, calls PAUSE, and then checks the completion bit again. At FWAITA, all four of the PP pointers are set according to the pointers in the FET; then FWAIT exits.

Entry Information

None.

Exit Information

The PP pointers updated.

Other Subroutines Called

PAUSE, which calls R.PAUSE.

Registers Destroyed

D.Z1 through D.Z5; also D.FIRST-1, which is not used by 1IU.

BACK

This backspaces the file one PRU. If the PRU number in the FNT entry is non-zero, it suffices to reduce it by 1 and exit. Otherwise, a stack request must be made. But if, at BACKA, it is found that bits 12-23 of the second word of the FNT entry are 0002B, the FNT entry is in rewound position where no further backspace can be performed, so control goes to DONEA. This cannot happen during the preliminary backspacing of the file except in the case that the backspace was called for before any of the file had been printed. If the same point is reached later in 1IU, because the page count requested was impossibly high, it is also true that the best thing to be done is to begin from the beginning of the file. So if the rewound position is reached, it is desirable to ensure that the buffer is empty before turning the file back to 1IR. At DONEA, the PP OUT pointer is set equal to the PP IN pointer. At DONE, the FET OUT pointer is set equal to the PP OUT pointer. The PP IN pointer already equals the FET IN pointer because it was set so after the last read (subroutine READ) and is not altered in the PP or the FET between reads.

If a stack request to backspace one PRU is necessary and possible, subroutine NODAT is called with 44B in the A-register; this code is inserted in the FET and the FNT entry and the stack request at MYRQ is partially set up. Then the PRU count in bits 24-41 of the second CM word of the request is set to 1. Then with function O.BPRU in the A register, subroutine DORQ is called to complete the stack request by inserting the function code, get it issued, and wait till it has been completed. Then exit from BACK.

Entry and Exit Information

None.

Other Subroutines Called

NODAT: DORQ, which calls R.EREQS and FWAIT, which calls PAUSE, which calls R.PAUSE.

Registers Destroyed

D.Z1 through D.Z5 directly. D.Z6 through D.T4 by the called subroutines. Also D.FIRST-1 by FWAIT.

READ

This is called to read one PRU into the buffer, which is 65 words long; to update the PP pointers accordingly; to store the length of the PRU in D.FNT+3; and to set D.FNT+4 to 20B if reading the PRU gave end of record/file status or zero otherwise. First set IN=OUT=FIRST in the FET. Then call subroutine NODAT with 12B in the A register, to insert this status in the FET and the FNT entry and partly set up the stack request. Then call subroutine DORQ with function O.READ in the A register, to complete the stack request by inserting the function code in it, get it issued, and wait until it is completed. Then reset the PP pointers according to the FET pointers. Then set D.FNT+3 and D.FNT+4 as noted above.

Entry Information

None.

Exit Information

The four buffer pointers in the PP are updated. D.FNT+3 contains the number of CM words in the PRU just read. D.FNT+4 contains 20B if reading the PRU gave end of record/file status, or zero otherwise.

Other Subroutines Called

NODAT: DORQ, which calls R.EREQS and FWAIT, which calls PAUSE, which calls R.PAUSE.

Registers Destroyed

D.Z1 through D.Z5 and D.T0 through D.T4 directly; D.Z6 and D.Z7 by NODAT. D.FIRST by FWAIT.

NODAT

This is called by READ and BACK with 12B or 44B respectively in the A register, to insert this status in the FET and the FNT entry, and to set up a stack request in MYRQ through MYRQ+9, all but the function code in MYRQ+3. However, for the call from BACK, the FWA field of the request has to be altered from FIRST to the PRU count (1) after NODAT has been called.

First get the first word of the FET from CM, insert the new status in its 5th byte, and return it. Then zero MYRQ through MYRQ+9, read the FNT entry to MYFNT through MYFNT+14, and insert the address of the second CM word of the FNT entry in bits 36-47 of the first CM word of the request (MYRQ+1). Then check the initial RBT pointer of the FNT entry. If this is 0 (impossible) go to PRERQA to use 1 as the physical unit number. Otherwise, read from CM cell P.RBR the memory size/100B (bits 0-11, in D.T2) and the RBR starting address (bits 36-59, in D.Z6 and D.Z7). Then read the first RBT word pair, get the RBR ordinal from bis 39-47 of its first CM word, and from the RBR starting address plus 38 times this ordinal read the first word of the RBR. Bits 42-47 of this word give the physical unit number, which we store in MYRQ+1, or if it is 0, use 1. Then copy the FIRST and LIMIT pointers into bits 24-41 and 0-17 respectively of the second CM word of the stack request. Then calculate the FET address from the FET number in bits 0-11 of the input register, and insert it in bits 42-59 of the second word of the stack request. Finally, set the level number in the FNT entry to 0, and copy 44B or 12B, which has been saved in D.Z1 since entry occurred at NODAT, into bits 0-11 of its third CM word. Then return the FNT entry to central memory and exit from NODAT.

Entry Information

The A register contains 44B for backspacing one PRU, or 12B for reading.

Exit Information

The FET and the FNT entry have been completely prepared for the stack request, and the request itself has been almost completely set up in MYRQ through MYRQ+9.

Other Subroutines Called

None.

Registers Destroyed

D.Z1, D.Z6 through D.T4.

DORQ

This is called, with function code O.BPRU or O.READ in the A register, when the stack request in MYRQ through MYRQ+9 is complete except for this code in MYRQ+1. It inserts the code there; then calls R.EREQS to issue the stack request; then calls subroutine FWAIT to wait until the file shows not busy; then exits.

Entry Information

The function code O.BPRU or O.READ in the A register.

Exit Information

The stack request has been completed, and the four PP pointers have been updated.

Other Subroutines Called

R.EREQS; FWAIT, which calls PAUSE, which calls R.PAUSE.

Registers Destroyed

D.T0 through D.T4 directly; D.Z1 through D.Z5 and D.FIRST-1 (which is not used by 1IU) by FWAIT.

CHAPTER 17 - TABLE OF CONTENTS

17.1	CIO - CIRCULAR INPUT/OUTPUT	17-1
17.2	1MT - LONG RECORD STRANGER TAPE DRIVER	17-2
17.3	1PE - TAPE WRITE PARITY ERROR RECOVERY ROUTINE	17-15
17.4	1RS - S-TAPE READ DRIVER	17-18
17.5	1RT - SCOPE AND X-TAPE READ DRIVER	17-23
17.6	1WI - TAPE WRITER DRIVER	17-24
17.7	1WS - TAPE WRITE DRIVER	17-28
17.8	1WX - WRITE EXTERNAL TAPES	17-36
17.9	2TB - BACKWARD POSITIONING ½" MAGNETIC TAPE	17-43
17.10	1TF - ½" FORWARD TAPE MOTION	17-44
17.11	4LB - LABEL READING/WRITING	17-46
17.12	4LC - Y-TAPE PROCESSOR	17-50

17.0 TAPE I/O

17.1 CIO

The following tape drivers 1RS, 1RT, 1WS, 1WI, 1WX, 1MT, 2TF, and 2TB are loaded directly by CIO in response to I/O requests issued on tape files. Refer to chapter 7 for a detailed analysis of the operation of CIO.

17.2 1MT - LONG RECORD STRANGER TAPE DRIVER

Purpose

1MT provides the capability to read and write variable length records which are larger than the 512 CM word maximum restriction of the S tape drivers. The size of any record may vary from the installation parameter IP.NOISE as a minimum to the size of the CM buffer as a maximum. In all other ways the use and function of the long record driver is the same as that of the S tape drivers. The L tape driver should be used only where records will be longer than 512 CM words, however, as 1MT operates slower than the S tape drivers.

Entry Information

1MT will be loaded whenever an L appears in a REQUEST function or on a REQUEST card. An indication that 1MT has been loaded can be found in the DEVICE TYPE field in either the FNT or FET. L tape format is specified when the top 2 bits of the right hand six bits are on, {i.e..100,000 11x xxx}. CIO will load 1MT in the same PP beginning at 1000.

Low Core at Entry Time

Location	Systext Name	Description
20-24	D.FNT	2nd word of FNT
25-31		3rd word of FNT
32-36	D.EST	1st word of EST
40-44	D.BA	1st word of FET
45	BS	Last buffer status in FNT
50-54	D.PPIRB	Input register
55	D.RA	CM Reference Address/100B
56	D.FL	CM Field Length/100B
57	D.FA	CM address of 2nd word of FNT
60-61	D.FIRST	Address of CM buffer pointers
62-63	D.IN	
64-65	D.OUT	
66-67	D.LIMIT	
74	D.CPAD	Control point address
75	D.PPIR	Address of input register
76	D.PPOR	Address of output register
77	D.PPMES1	Address of 1st word of PP message buffer

Exit Information

For a normal exit 1MT will update the PRU count in the FNT, mark the function code complete in the FNT and FET and update the CM circular buffer pointers.

If a physical tape mark and/or the end of reel reflective spot is encountered on any of the read functions 1MT sets the end of information and/or the end of reel bit in D.FNT+9 and calls 3RP. 3RP will decide whether to return to the user or to 1MT. See 3RP for detailed description.

If on a write the end of reel is encountered and the UP bit is off 1MT calls CLO with a 370 function (close Reel Unload) in another PP and makes a normal return to the user. If the UP bit is on, however, 1MT returns to the user program without calling Close Reel Unload.

An exit without updating the CM buffer pointers and with the appropriate error flags set will be made for the following situations:

1. Unrecoverable parity error
2. Device capacity exceeded
3. Illegal function code
4. Error bits set during a storage move
5. FET word 7 or LIMIT not within field length following a storage move.

General Logic

WRITE OPERATIONS

The 1MT I/O method differs greatly from that of the other tape drivers. 1MT cannot use a PP buffer but must transfer data directly to or from central memory. For a write the S tape driver would transfer the data between OUT and IN to its PP buffer, calculate the number of PP words contained in that block, then issue one OAM to write the entire block to tape in one operation. 1MT cannot do this as one tape block may be larger than the available PP space or even larger than the PP itself. Therefore, on a write, it transfers each record one CM word at a time from the CP to the PP and then one or two bytes at a time from the PP to tape.

The method used for write operations is described first. For WRITE or WPHR the PRU size is the number of words between and including OUT to IN-1. Each write causes the

entire buffer to be written out. For WRITEN the PRU size is defined in the record header word. In either case this length can be modified by the unused bit count so that the last word in the record is a partial CM word. The PRU size is compared to the MLRS field in the FET and if it is less than or equal to MLRS the write continues, otherwise the job is terminated and the device capacity exceeded flag is set.

If buffer wrap around can occur the write is done in two sections. The CM buffer is considered in two segments; the data contained between OUT and LIMIT-1 and that contained between FIRST and IN-1. If a partial CM word is to be written there is the possibility of a third write section. The write falls through each section exhausting the word count for that buffer segment before going to the next section. The jump instruction at the end of each section is modifed to cause a jump either to the next section or to the terminating procedure .

In each write section a central memory word is read into PP locations D.T0 through D.T4 and then is output one or two bytes at a time. Between each CM word the CM address of the location of the next word to write is incremented, the word count for this segment is decremented and compared to zero; and the channel is checked for empty to insure that bytes are being transmitted at the expected rate.

Each write loop may be left in three ways:

1. If the channel remains full after each 5 bytes are output. This condition means that bytes are no longer being removed from the channel buffer. (The unit could have dropped ready during the operation or some other equipment malfunction could have occured). In any case the record is bad and must be rewritten. The channel is disconnected, WEO (wait end of operation) is called, the tape is backspaced and the write is retried.

2. Following each OAM the A register is checked to determine if it went to zero. If it didn't the bytes in question have not been written out. The unit may have accidentally been cleared or some hardware malfunction may have occurred. The record should be rewritten, therefore, an exit from the write loop is made.

3. The word count for this segment is exhausted. If there is wrap around a jump to the second write loop is made and the above process continues until the second word count becomes negative, channel buffer remains full, or the A register does not go to zero. If wrap around does not occur or at the conclusion of the second write loop a jump is made to the section which outputs the remaining bytes of the last CM word.

When the entire record has been output error checking begins. The channel is disconnected and WEO is called which checks for transmission parity errors or unit not ready during the last operation. If either condition is found, the lost data bit in the status word is set. This does not mean true lost data occurred but is used as a flag to WTT to cause the write to be reattempted. Finally, if there were no errors, WEO pauses waiting until end of operation and then returns to WTT.

Upon return to WTT is lost data status is found the tape is backspaced and the record rewritten. If the lost data flag is not set, WTT next looks for parity errors. If they exist the write is reattempted up to ten times. At that time the parity error routine PAR is called. If the EP bit is on the message PARITY ERROR is written in the dayfile, the parity error flag in the code and status field of the FET and FNT is set, and a return to the user program is made. If the EP bit is off the same message is posted at both the control and the dayfile. The operator then has the option to type either n.GO which causes the job to terminate without error bits set or to type n.DROP. If parity errors were encountered while trying to write a record but a successful write was made the message WPE RECOVERED is written in the dayfile.

When a successful write has been accomplished the PRU count is incremented and the new OUT and IN are written in the FET. If the request is a WRITE or a WPHR a return to the user program is made. If the request is a WRITEN another record is written and the above process continues until OUT=IN.

Switches that need attention are mostly concerned with BCD/DC conversion. If the file is BCD and a 6681 is in use the central memory buffer must be converted from DC to BCD before the record is written. Conversion is done by reading blocks of the record into the PP, converting it, and writing it back to the central memory buffer. The buffer must also be restored to its original form so it is converted from BCD back to DC after it is written to tape. {Herein lies the speed problem for 1MT} The parameters needed for CNVRT are calculated and saved. CNVRT is called first before the record is written and again at the end of the write as though the operation were a read. For each record the switches must be set to first convert from DC to BCD and then to convert BCD back to DC as though the request were a read.

READ OPERATIONS

The circular buffer parameters IN, OUT, FIRST, and LIMIT have been checked for validity in CIO before entering 1MT, and are now used to determine the maximum PRU size for the

next read. For a READN request the new OUT is read from the FET before processing each record. For READ, RPHR and READN the available buffer space, which is IN to OUT-1, must be greater than or equal to MLRS or the read is not attempted. This restriction does not apply to READSKP as the entire record is not required to fit in the circular buffer. At the conclusion of the READSKP, however, the total record length passed over is compared to MLRS. It must be less than or equal to MLRS or an error is returned.

As in the write procedure the read is divided into two possible segments dependent upon wrap around. Unlike the write it cannot be determined until the end of record is hit that buffer wrap around actually occurred. It is necessary to provide for the possibility. The first section is from IN to LIMIT-1 for READ, RPHR or READSKP and from IN+1 to LIMIT-1 for READN. If wrap around cannot possibly occur only one section is used which begins at IN (or IN+1) and ends at OUT-1.

The absolute central memory address of the first word address of the second section for the conversion routine is not stored until after the record has been read when it can be determined that wrap around actually occurred. If wrap around is possible the jump instruction at the end of the first segment is modified to cause a jump to the second section rather than to the terminating procedure.

The read loop inputs 12 bit bytes one or two at a time and stores them beginning in D.T0 through D.T4. When 5 bytes have accumulated they are written to the CM address stored in line in each read section. Between each CM word the following book keeping must occur: the CM address of the location in the buffer for the next word must be incremented; the word count must be decremented and compared to zero. Between each PP byte a check for end of record is made. This is necessary for a tape containing other than full 60 bit words. All of the above checks must be done within the time limits discussed in the section "I/O Timing Constraints".

There are two exits from the read loop. The first occurs when the word count is exhausted (goes negative) and the second occurs when the end of record is hit. If the last CM word is partial the remaining bytes are filled with binary zeros and the byte count indicator is set to the number of 12 bit bytes of zero that were needed. This count is later multiplied by twelve and returned as the unused bit count. If the read finished before the word count went negative a flag is set to indicate the read finished in the buffer. If the word count went negative in the last possible read segment buffer overflow occurred. IN=OUT-1 and there is not

enough room in the CM buffer for the rest of the record. The remaining words are flushed and the record length indicator is incremented once for every 5 bytes and once for the last partial CM word should it exist. This is necessary to calculate if the record on tape was larger than MLRS since this check cannot be made for a READSKP before the operation begins. If on any request the record was larger than MLRS an exit is made with the device capacity exceeded error flag set.

If the record just read was less than or equal to the installation parameter IP.NOISE the record is flushed as noise and the read is reattempted.

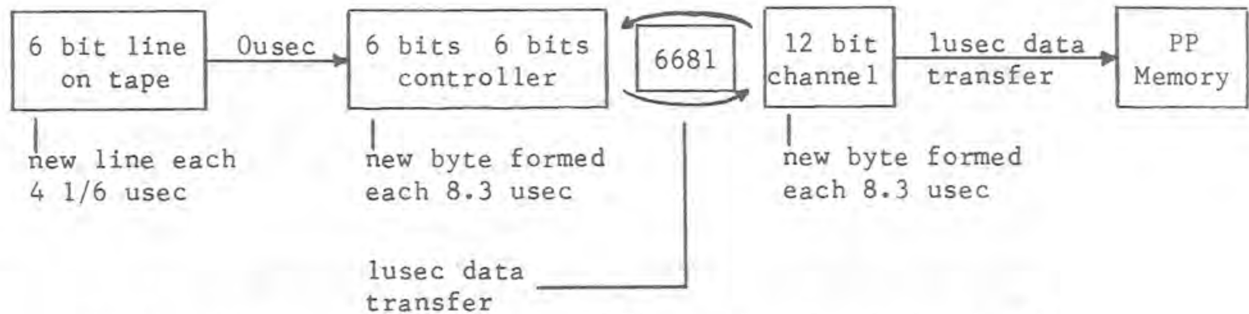
When end of record is hit WEO {wait end of operation} is called which calls STS to obtain unit status then loops until end of operation. RDT later checks the status word for lost data or for parity errors. If lost data is found the read is reattempted until it is successful. If parity errors are found the read is tried up to nine times. If it is still unsuccessful the parity error routine PAR is called. If the EP bit is on the parity error flag is set in the code and status field, the message READ PARITY ERROR is displayed in the dayfile only and a return is made to the user program. If the EP bit is off the message is displayed at both the control point and in the dayfile and a pause is made waiting for the operator to type either n.GO or n.DROP.

If there were no errors on the last attempt the conversion routine is called when need and the PRU count is incremented. If the request was a READN the record length in CM words and the unused bit count are returned to the data header word which is the previous IN. If the request is not a READN the unused bit count is returned to word 7 of the FET. For all requests the new IN is written in the FET and for RPHR the new OUT must also be returned as it was set to IN before the read started. If the request is a READN an attempt is made to read the next record, otherwise a normal return to the user program is taken.

Timing Constraints

The following situation is for a read. The underlying principle of 1MT is to transfer words from tape to central memory at the same speed that the tape is moving. This is accomplished by allowing bytes to remain in the PP only long enough to assemble one central memory word. The following diagram should clarify the critical timing dependent situation.

SCOPE



16.6 usec max time between 12 bit bytes entering PP at 800 bpi.

Updating of necessary parameters must be done between PP bytes, while each CM word is being assembled. The key facotr is than an IAM must be executed every 16.6 usec (800 bpi).

The channel buffer must be empty when the controller is ready to transfer the next 1/2 byte. Similarly the controller must continue transferring data to the channel at the same rate it is receiving data from the tape. If the PP falls behind and fails to remove data from the channel at least every 16.6 usec lost data occurs.

The most likely possibility for failure occurs when the PP tries to write the data word to central memory. If the PP cannot gain access to the write pyramid it will be unable to get back to the IAM instruction in time to receive the first byte of the next word.

The problem in the case of a write is similar. The PP must continue to supply the channel with data at the same rate the tape is moving. Here again the conflict comes at the read pyramid. If the PP should be forced to wait for data from central memory, the channel also will not receive information. After a wait equal to the time necessary to output 2 1/2 characters the end of record procedure is initiated.

For these reasons timing must be carefully considered before changing code in the central read or write loops.

SUBROUTINES

CCH Close Channel

CCH calls R.DCH which causes the channel to be dropped.

CLECT   Selects 6681 or 6684 mode

CLECT looks at the EST then selects the proper mode. If the file is BCD and a 6681 is present switches are set to cause the convert routine (CNVRT) to be called.

CEOR   Check End of Reel

CEOR loads the A register with bits 9-13 of the code and status field. These bits contain the end of reel indicator as well as other error flags.

CKLP   Check Load Point

If the PRU count is greater than zero an immediate return to the calling program is taken. Otherwise the tape is at load point and OPE function 340 is called to position the tape correctly. OPE will leave the tape ready to read the first data record. CKLP loops monitoring the PRU count. In the process it calls CEOR to check the code and status field for errors. If errors are found an exit from 1MT through NORM is made. Otherwise a normal return is taken when the PRU count becomes positive zero.

CKR   Check Ready

CKR checks for unit ready. If the unit is not ready the message MTXX,CHXX,NOT READY is written at the control point and a pause is made until the operator readies the tape or drops the job.

If the function is a write CKR also checks for a write ring. If it is not present the message MTXX,CHXX,NO WRITE ENABLE is displayed until a ring is inserted or the job is dropped.

CLEAR   Issue Clear and Disconnect

Issues a clear and disconnect. Checks for converter errors but does not call subroutine STS where the unit status would zero out the status word, ST, which is necessary so that the last status is available when checking for end of reel in RCRU.

CON Connect

CON reads D.EST+4 and connects either the 6681 or 6684. If for any reason the converter does not accept it, the connect is retied until successful.

ERR Error Message

ERR is called with the address of an error message in the A register. The message is written in the dayfile only using R.DFM.

FCN Function

FNC issues function codes. The desired code is loaded in the A register and FCN is called. If the function code is not accepted for any reason the appropriate message is written and the process is retried until successful.

ILEG Illegal Function Code

ILEG is called when an illegal or non-existent function code is found. ILEG calls ERR to write the message, 1MT ARG ERROR, and if the EP bit is on, sets the illegal function code bits in the code and status field.

MSG Message

MSG is entered with the address of a message to be written in the A register. This becomes the second line of the message which is to be written at the control point only.

MUCH Mask Unit and Channel Number into Message

MUCH is called to mask the unit and channel number into the first line of the two line message.

```
i.e.    MTXX CHXX
    NO WRITE ENABLE.
```

It is done separately from MSG so that the message may be sent to either the dayfile or the control point.

OCH Open Channel

OCH gets the channel number from the EST entry and if it is different from the primary channel number calls R.STB to mask it into the channel table. Next the 6681/6684 select code is issued and the response is checked. If the converter does not respond within a given time period the channel is disconnected and the message MTXX,CHXX,NO RESPONSE is written. A pause is made, the EST entry reread and the code is issued once more. This continues until a successful select is made.

PAUSE Release Equip, Pause, Reconnect

PAUSE through a series of other subroutines closes and releases the channel pauses for relocation and finally opens the channel and reconnects.

PREP Prepare for I/O

PREP is a series of calls to other subroutines. The series, OCH,CON,CKR,CLECT when combined make a tape ready for I/O.

PRL Pause for Relocation

PRL goes first to R.PAUSE where it waits until a storage move is completed. Upon return it stores the new RA and checks for errors. If the field length was decreased PRL determines if LIMIT and the seventh word of the FET are still within the new field length. If any errors are found an exit from 1MT is made. Otherwise the new field length is stored in D.FL and a normal return is taken.

RCRU Call Close Reel Unload

RCRU is called only for the write operations when the reflective spot has been found and the UP bit is off CLO is called into another PP with 370 function. Whtn the end of reel bit is turned off in the FNT a normal return is made. If the UP bit is on, however, a return to the user is made.

STS Status

STS requests 6681/6684 and unit status. It leaves the 6681/6684 status in Temp 1 and bits 18-12 of the A

register. When if the end of reel bit {EOR is turned off a normal exit from 1MT is taken.

<u>WEO</u> Wait for End of Reel

WEO is called at the end of an operation. It calls STS and checks for transmission parity errors or unit not ready. Either could have occurred during the operation. If either is found a flag {lost data} is put in ST and a return to the calling routine is made where the appropriate action will be taken. If there were no errors WEO loops waiting for the end of operation status to be set.

<u>XPE</u> Transmission Parity Error

First a delay loop is entered which waits for the slowest equipment on the channel to complete its operation. Then a master clear is issued and calls to CCH, OCH, and CON are made. The effect is to clear the channel drop the equipment, pause for relocation, and, finally, reconnect.

<u>ERROR MESSAGES</u>

The following are two line messages issued by 1MT. The first line is MTXX CHXX, and the second is listed below.

<u>EQUIPMENT REJECT</u>

Written by CON after a connect code has been issued and rejected by the 6681. The possibilities of unit not ready, unit reserved and transmission parity error have been checked and found negative. A master clear is done and the connect is retried.

<u>NO RESPONSE</u>

Written by OCH when 6681 select code has been issued but no response was made. The select is retried.

<u>NOT READY</u>

Output from either CKR of CON when the requested unit is found not ready. Pauses while operator makes unit ready. Processing resumes when operator readies unit.

NO WRITE ENABLE

Written by CKR when a write has been issued and the tape on the requested unit does not have a write ring. When operator inserts ring processing resumes.

READ PARITY ERROR

Written by PAR after 10 unsuccessful attempts to read a record. When the message is displayed the block of words containing the parity error is in the central memory buffer, but the buffer parameters have not been updated. Operator may type either N.GO which causes the job to resume processing the next record, or N.DROP which terminates the job.

RESERVED

Written by CON when a connect is attempted to a unit reserved by another computer. The connect is reattempted until the job is dropped.

XMSN PARITY ERROR

Issued by CON, FCN, or PAUSE whenever a connect code or function code is rejected because of a transmission parity error. A master clear is done and the code is issued again.

WPE RECOVERED

Written by PAR whenever a successful write has been accomplished after parity errors were found during earlier attempts.

PARITY ERROR

Written by PAR after 10 unsuccessful writes have been attempted. If EP bit is off execution is continued when N.GO is typed, or terminated when N.DROP is typed.

The following are one line messages issued by 1MT.

1MT ARG ERROR

Issued by ILEG after comparing the operation code to all the accepted function codes without finding a match. If the EP bit is off abort the control point.

FET LESS THAN 7 WORDS

The FET length parameter in word 2 of the FET is less than 2. A 7 word FET is required for 1MT.

MLRS CHANGED to MAX BUFFER SIZE

The MLRS field was zero and has been changed to either LIMIT - FIRST - 2 for READN or WRITEN or the LIMIT - FIRST - 1 for the standard Reads and Writes and the operation is completed.

RECORD SIZE GT MLRS

The record just read was longer than MLRS or the record length request in the data header word for a WRITEN is larger than MLRS.

REQUESTED RECORD SIZE=0

The record length requested in the data header word for a WRITEN is zero.

REQUESTED RECORD SIZE=NOISE

The requested record size in the data header word for a WRITEN is less than or equal to IP.NOISE.

UNUSED BIT COUNT GT 59

The unused bit count in either word 7 of the FET for standard writes or in the data header word for WRITEN is less than 59.

17.3 1PE - TAPE WRITE PARITY ERROR RECOVERY ROUTINE

General Description

1PE is the tape write parity error recovery routine used in conjunction with 1WI, 1WX, and 1WS, to recover write parity errors on Scope, X, and S tapes. 1PE is loaded on top of the calling driver and has a copy of the record to be recovered beginning around 2777B of PP memory. If the record to be recovered is in BCD, the calling driver will have previously converted it to internal BCD.

Communications

The chief communications between 1PE and the calling tape write driver is via the following direct cell contents:

1. 1PE scans these cells upon entry

   D.SV1 = FWA of data in PP memory

   D.SV2 = Data byte count

   D.SV3 = CM word count used to update the "OUT" pointer (X-tapes only).

2. 1PE sets these cells on return to the driver

   D.FR6 = Converter status

   D.FR7 = Equipment status (including end-of-reel if detected during recovery)

   D.SV1 = FWA of data in PP buffer (unless a zero length PRU of level 17 was recovered, in which case D.SV1 is set to 17.)

   D.SV2 = Data byte count

   D.VS3 = CM word count used to update the "OUT" pointer

3. The following direct cells must never be altered by 1PE if it is going to return control back to the calling driver.

   D.FIRST
   D.IN
   D.OUT
   D.LIMIT

4. The following direct cells must be re-initialized at 1PE entry time and be current upon return to the calling driver.

   D.FNT through D.FNT+4
   D.EST through D.EST+4

Tape Positioning Philosophy

When 1PE receives control from a tape write driver, the tape will be positioned immediately after the bad record. At this point a reverse read is issued and the number of bytes that are sent down the channel are counted. The count thus obtained is compared with the actual byte count (D.SV2) of the record written. There are three possibilities, read count greater than data length, read count equal to data length, and read count less than data length.

CASE I Read count greater than data length.

This condition may be caused by excess skew on the frames of the record just written. It could also be caused by noise (incomplete or unsuccessful erasure) in the inter-record gap between the last good record written and the record that incurred the parity error. It could, of course, be caused by a combination of the above. If the CASE I condition is caused by noise in the gap (CIG), then there is no way for the software to know whether or not the reverse read has "carried back" into any or all of the last good record. Accordingly, the message "MT XX FILE POSITION UNCERTAIN" is posted and a pause for operator decision then follows. A "GO" response is acknowledgment of the fact that the last good record may be damaged in the continuation of the recovery attempt that follows as a consequence of the operator "GO" decision. If the integrity of the file is at all critical, it is suggested that a "GO" never be given when the above message is encountered, but that the job be dropped and re-run using another reel of tape.

CASE II Read count equal to data length

If the read count is exactly equal to the data length, then the file is assumed to be positioned correctly in front of the bad record. There is actually a "tolerance" associated with this check such that if the read count is within 6 bytes (12 frames) of the data length, it is assumed to be close enough to be properly positioned in front of the bad record. The causes of a low reverse read byte count and the action taken if it is not within 6 bytes of the data length is discussed under CASE III.

CASE III Read count less than data length

There are several possible reasons for obtaining a low byte count on the reverse read. As a result of edge damage, oxide damage or dirt, an occasional frame may not be detected by the read head. Thus, the entire record could have been reverse read and many frames just not detected (the data will appear shifted together where the frames were missed) by the read head. If only a few frames are dropped in the above manner, the read byte count will still be within 6 bytes of the data length and it is safe to continue recovery, i.e., the tape has been correctly positioned before the bad record.

If two or more consecutive frames are not detected by the read heads, the transport sends an end of record signal and stops the tape motion. If this happens during the first part or middle of the reverse read, 1PE cannot be certain that the entire record will be positioned in front of the erase head (for the ensuing skip bad spot). Accordingly, the message "MT XX POSSIBLE RECORD FRAGMENT" is published when the byte count is not within 6 of the data length. A "GO" response to this message implies acceptance of the possibility that a portion of the bad record might still exist on the tape. The "GO" causes a continuation of the recovery procedure.

Re-Write of Bad Record

After the tape has been successfully positioned (or a "GO" was given to either "MT XX FILE POSITION UNCERTAIN" or "MT XX POSSIBLE RECORD FRAGMENT") 1PE issues a skip bad spot and then re-writes the record. If the parity on the write is now good, control passes to a verification (see below) procedure. If the parity is bad, control will pass to the positioning algorithm as long as the retry count has not been exhausted. For each retry, therefore, another skip bad spot is issued causing successive re-writes to take place farther along the tape.

Record Validation

After the record has been re-written in good parity all data in the record is verified. To do this, the record is reverse read and the byte count read is compared to the data length. If they are equal and if every byte of the data read back matches every byte of the data written, then a forward read is issued. Again, the byte count must be exactly correct, the data must compare exactly, and parity on the forward read must be good. If all of the above conditions are met, the message "MT XX WPE RECOVERED" is posted and the record is considered recovered. If any one of the above conditions is not satisfied, control will go to the positioning algorithm for another retry. If the retrys are all used up and the EP bit is not set, the message "MT XX WPE UNRECOVERED" is issued and operator action is required (GO or DROP). If the EP bit is on, the parity error flag is set in the last code and status and control returned to the user.

17.4 1RS

General Description

1RS reads one or more records in S tape format (a logical record is a physical tape block) from half inch seven track magnetic tape. No special significance is attached to any character or series of characters so that the data transferred to the circular buffer accurately reflects what was read from tape. For coded files, of course, the data is converted into display code. The only data structure limitations are hardware imposed and consist of limiting the read "resolution" to the nearest byte (2 characters) and record size to 5120 characters. The maximum record size of 5120 characters is a consequence of PP buffer size limitations and applies to both binary and coded records.

Functions Recognized by 1RS

1RS recognizes the following functions: READN, READ, and READSKP. Each function is described in detail in the following paragraphs.

READN Request

The READN request is the only means of executing high speed data transfer reading with an S-tape. It allows the central processor to dynamically remove data from its circular buffer while the read driver simultaneously transfers data from the tape into the circular buffer. If the user program is removing data from the circular buffer at a sufficiently high rate (at least as fast as the read driver is filling the circular buffer) the READN can continue until end of reel or some error condition is detected.

The user interfaces with 1RS via his FET and his circular buffer during a READN request.

A. Utilization of FET word seven

During initiation of the READN request 1RS scans the MLRS field in FET word seven to obtain the size of the maximum allowable record for this particular READN request. If a record greater than MLRS words is detected during the non-stop read, the read will be terminated with device capacity exceeded status, and no data from the excessively large record will be transferred to the circular buffer. If the MLRS field is zero, 1RS will set it to device capacity (512_{10} CM words) and assume this value for execution. If the MLRS field exceeds 512, it is set to 512 and that value is again assumed for execution. Thus, for a READN request, FET word seven is accessed only at the start of the operation, and will only be written into if MLRS is zero or an invalid quantity.

B. FET word one

The first word of the FET is used in the normal way; with the code and status updated on completion of the READN request.

C. FET IN and OUT pointers.

Each time a record is read from tape and transferred to the circular buffer, the IN pointer is updated and the OUT pointer is re-read from the FET. Non-stop reading will continue as long as there is enough room to insert the next record (unless an error is encountered).

D. Circular buffer

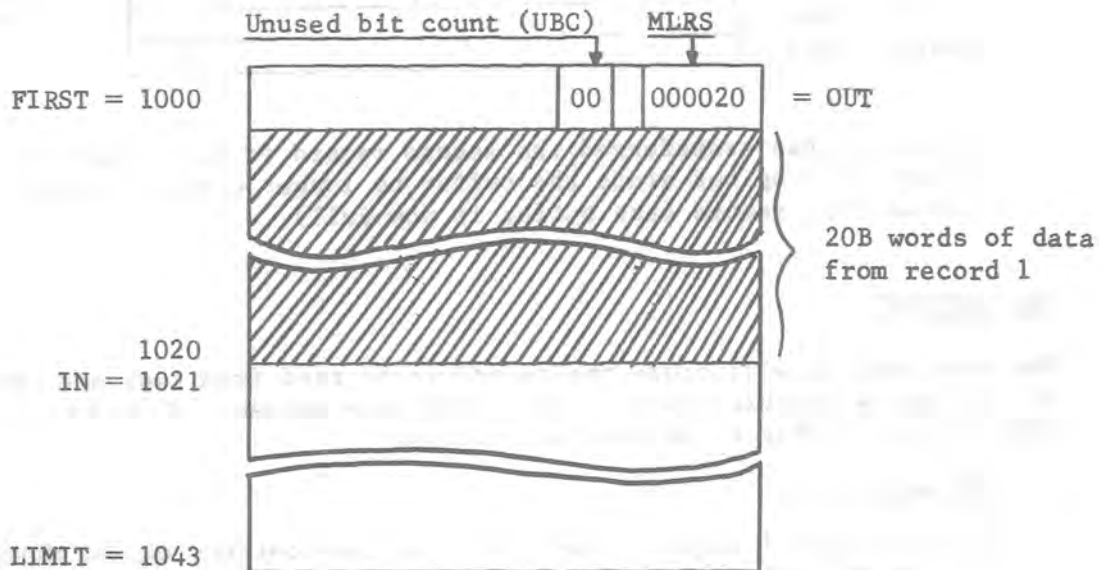
With each record read from tape, 1RS constructs a header word containing a central memory word count and an unused bit count. The unused bit count denotes the number of unused bits (this will be a multiple of 12_{10}) in the last word of the record. The header word is then appended to the beginning of the record data and the header and record are transmitted to the circular buffer. For example, suppose a user issues a READN request under the following conditions:

FET word 7, MLRS = 20B
FIRST = IN = OUT = 1000B
LIMIT = 1043B

1RS will initiate tape reading and suppose, for the sake of illustration, the first record is exactly 20B CM words in length. The circular buffer and FET will then look like this:

FET word 7, unchanged
FIRST = OUT = 1000B
IN = 1021B
LIMIT = 1043B

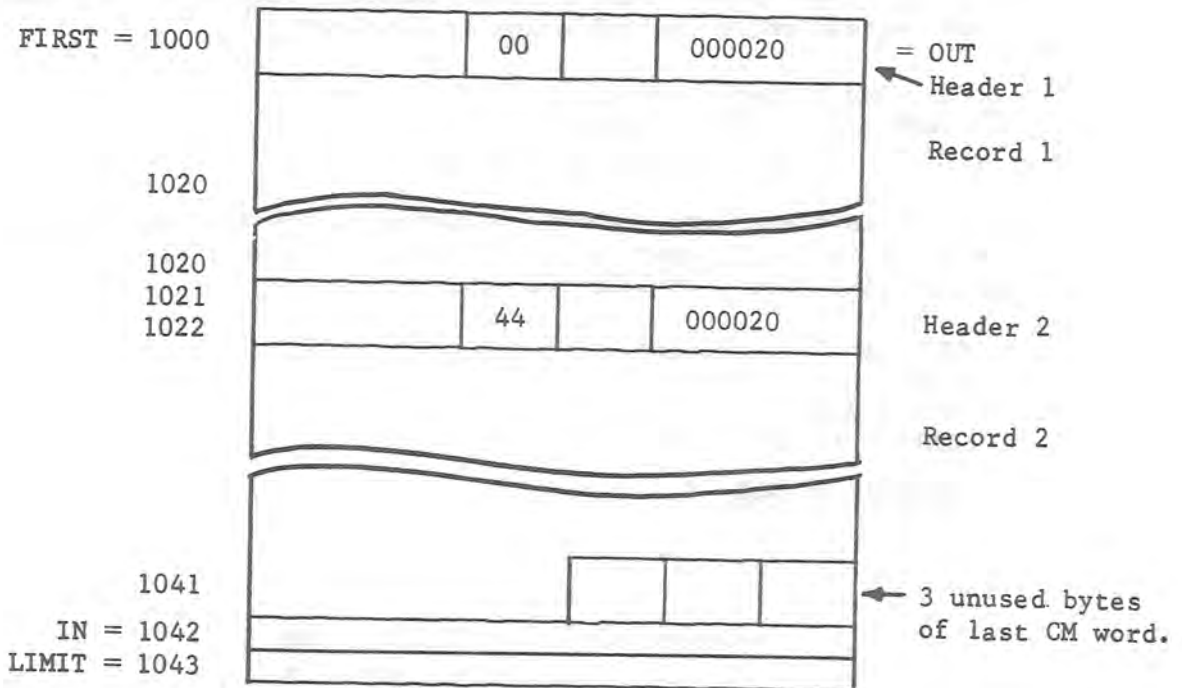
CIRCULAR BUFFER



Since there is room for another record (MLRS words plus one word for the header), 1RS will continue non-stop tape motion and read the next record. Suppose that this record is 17B full CM words plus 2 bytes in length. After reading this record, we have:

FET word 7, unchanged
FIRST = OUT = 1000B (Assuming user hasn't removed data from the
IN = 1042B circular buffer)
LIMIT = 1043

CIRCULAR BUFFER



After 1RS has transferred the second record it will complete the request and drop out since the buffer no longer contains enough room for another record (the buffer is now full).

READ REQUEST

The READ request will cause one record to be read from tape and transferred to the user's circular buffer. The interface between 1RS and the user's FET/circular buffer is defined as follows.

A. FET word seven

As in the READN request, the READ uses the contents of the MLRS field for the maximum allowable record size. If the record read from tape exceeds MLRS words, no data is transmitted to the circular buffer and

device capacity exceeded status is returned to the user. Also, if MLRS is zero or invalid, it will be set to device capacity (512 CM words) and device capacity used for execution. The difference in usage of FET word seven between a READ and a READN request is that a READ will set the unused bit count field (UBC) if the record read is not a full CM word. This is because there is no header word attached to the record and returned to the user's circular buffer.

B. Circular Buffer

The record read from tape is transferred to the circular buffer and the IN pointer is advanced to show how much data (to the nearest CM word) was read from tape. If the last CM word of the record is not entirely data, the number of unused low order bits of that word will be reflected by the UBC field in FET word seven. The READ request causes only one record to be read from tape.

READSKP REQUEST

The READSKP differs from the READ request in the following ways:

A. If the circular buffer has less space available than is required for the record read, the buffer will be filled and the excess data discarded.

B. If the level parameter associated with the READSKP request is equal to 17B, then one record is read from tape and the file is skipped over until end of file is detected.

INTERRECORD GAP UTILIZATION

Below is a simplified description of the scheme employed by 1RS to take advantage of interrecord gap time (5 ms at 150 IPS). This is the non-stop (READN) loop.

Step 1. Check for room for 1 record (MLRS words + 1).

Step 2. Begin tape motion.

Step 3. Read a record from tape to PP buffer.

Step 4. If BCD/6681, convert it to display code.

Step 5. Check parity (assume good parity).

Step 6. Space for next record (assume yes).

Step 7. Re-activate channel (continue tape motion).

Step 8. Transfer record from PP to circular buffer.

Step 9. Go to step 3.

BCD TRANSFER RATES

When a 6684 data channel converter is available, BCD read rates will be the same as binary. In the absence of a 6684, conversion from internal BCD to display code must be done by a software algorithm within the PP. This algorithm converts forty per cent of the record as it comes into the PP from the data channel. The remaining sixty per cent is converted in the PP and the entirely converted record is then sent to the user's circular buffer. For each five bytes read from tape, bytes 1 and 3 are converted "on the fly" and bytes 2, 4 and 5 converted while waiting for end-of-operation.

TAPE MARK AND END OF REEL PROCESSING

1RS unconditionally calls in 3RP when a tape mark is detected. 3RP will be loaded into the same PP and will check the end of information bit in direct cell D.FNT+9 - the bit set by 1RS when a tape mark is detected.

When end of reel is detected 1RS will set the end of reel bit in D.FNT+9 and call 3RP if the UP bit is not set. See IMS write up for 3RP concerning the action taken in these conditions.

READ PARITY ERROR RECOVERY

Read parity error processing is completely contained within 1RS. It consists of a number of retrys (backspace/re-read) at the bad spot on tape. If these are unsuccessful, the final three attempts will back up over the bad record and the two previous good records and try to re-read the bad spot "on the fly". In the event recovery is not possible, the standard EP bit conventions will apply. Specifically, an operator GO/DROP is solicited if the EP bit is off, and the parity error status returned to the user along with control, if the EP bit is set. Also, with the EP bit on, the loaction pointed to by the updated "IN" pointer will contain the user's previous "IN" pointer. The area of suspicious data in the circular buffer is thus delineated.

7.5 1RT

The IMS for this will be supplied at a later time.

17.6 1WI - TAPE WRITER DRIVER

General Description

This driver writes one or more physical record units (PRUs) from a circular buffer in central memory to half-inch magnetic tape in SCOPE Standard format. It recognizes the following request codes: WRITE, WPHR, WRITER and WRITEF. A brief discussion of the action taken by 1WI for each of these request codes may be found immediately after the statement concerning Principles of Operation.

Principles of Operation

The most significant technique for throughput improvement used by 1WI applies to both binary and coded writes and specifically is the efficient utilization of inter-record gap time. Assuming that the circular buffer contains N PRUs where N is treater than or equal to 2, the driver will enter its high speed routine and process PRUs in the following manner.

Step 1 Read PRU 1 from CM into PP memory.

Step 2 If BCD and no 6684, convert the first 116 CM words of the 128 word PRU. If binary or BCD with a 6684, go to next step.

Step 3 Begin tape motion.

Step 4 If binary or BCD with a 6684, go to next step. If BCD without 6684, use the tape start-up time to convert the remaining 12 words of the PRU - an assumption is made at this point namely that the start-up time will never be less than 2.0 ms.

Step 5 Write the PRU in the PP buffer to tape.

Step 6 Read next PRU from CM into PP memory.

Step 7 Same as Step 2. above.

Step 8 Wait for end-of-operation to be returned by the equipment for the previous PRU.

Step 9 Continue if not end-of-tape or parity error. Perform E-O-R procedures for end-of-tape and 1PE processing for parity errors.

Step 10 Begin tape motion (or continue non-stop tape motion as the case may be).

Step 11 Go to step 4.

The above logic flow, while only a summary, indicates how the driver uses inter-record gap time while in a "non-stop" or high speed mode of operation.

If the user program is putting data in its circular buffer at a rate approximately equal to the rate at which 1WI is removing data from the buffer, the driver may enter its low speed routine. This will happen if at any given instant, 1WI detects only one PRU in the buffer. In the low speed routine the driver does not drop out of the PP, but it does allow tape motion to cease between PRUs.

A second technique for obtaining increased throughput is used in the BCD converion procedure, which is utilized in the absence of a 6684. Since a PRU is always a multiple of five PP bytes (short PRUs are discussed later) a series of five macros are used, one for each byte position of a CM word. The macro that converts the low order byte of a CM word checks for a line terminator before converting. These five macros are executed serially - in - line thereby reducing the number of passes through the convert loop. In addition, two conversion tables are used by each macro.

In the case of a short PRU, the byte count will always be 4 modulo 5. In order to use the above convert macros, the byte count is incremented by one to make it 0 modulo 5, a switch is set (SW11) and the convert performed. SW11, when set to indicate a short PRU, will decrement the byte count just before the OAM instruction is issued, thereby causing the correct number of bytes to be written to tape.

WRITE REQUEST

The WRITE request is the only one in which the central processor program can dynamically transfer data from the circular buffer (by placing data into the circular buffer and advancing the IN pointer while the driver simultaneously removes data, writes it to tape and updates the OUT pointer). Thus, the WRITE request is the only vehicle for sustained high speed writing on SCOPE Standard Tapes.

If a WRITE is issued with less than a full PRU of data in the circular buffer (128 words for BCD and 512 words for binary), 1WI will return to the user. If there is at least one full PRU in the circular buffer, 1WI will continue writing until the circular buffer contains less than a full PRU. The user's OUT pointer is updated after each PRU and the IN pointer read in to re-compute the amount of data in the circular buffer.

WPHR REQUEST

The WPHR request causes a single PRU to be written in either binary or coded mode. On coded requests only conversion from internal to external BCD is performed. If the circular buffer contains 512 words or less, the buffer is emptied and the OUT pointer is set equal to the IN pointer. If the circular buffer contains more than 512 words, the OUT pointer is advanced to show that 512 words were removed from the buffer and written to tape and, additionally, the device capacity exceeded flag (10_8) is returned in the status field of the FET.

WRITER REQUEST

Data is written from the circular buffer to tape in PRUs until the buffer is empty (a short PRU written). A level number obtained from bits 14-17 of word three of the FNT is appended to the short PRU. If the buffer contains an integral multiple of PRUs, 1WI will terminate the WRITER request by writing a zero length PRU of the specified level. In either case, the OUT pointer is set equal to the IN pointer and the level number or level mark is composed of eight characters having the following appearance:

	Level Number	External BCD Representation			
CODED FILES:	0	2020	2020	2020	2020
	1	2020	2020	2020	2001
	2	2020	2020	2020	2002
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.
	12	2020	2020	2020	2012*
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.
	17	2020	2020	2020	2017
BINARY FILES:	0	0000	0000	0000	0000
	1	0000	0000	0000	0001
	.	.	.	.	.
	.	.	.	.	.
	.	.	.	.	.
	17	0000	0000	0000	0017

WRITEF REQUEST

The WRITEF function causes a logical end-of-file to be written on tape. A logical end-of-file is a zero length PRU of level 17. If any data is in the circular buffer it is first written and terminated with a level zero short or zero length PRU. This terminates the logical record. If the buffer is empty and the last request was a WRITE, a level zero is written denoting logical end-of-record, followed by a level 17 to denote logical end-of-file.

LOAD POINT

In order to avoid writing on unexpired labelled tapes, 1WI calls OPEN REEL when the PRU count is negative. No writing will be done on the tape until the PRU count goes to zero, indicating that writing is permitted.

PARITY ERROR RECOVERY

While operating in the high speed routine, by the time the status for a given PRU has been checked for parity, the next PRU to be written has been read into the PP buffer. Hence, when a parity error is encountered, the PRU sustaining the error must be re-read from CM into the PP buffer. If the PRU in error is in BCD mode, the PRU is converted. This is done even if the driver has been using a 6684 (no software conversion necessary). At this point 1WI is ready to call 1PE in "on top" of itself to recover the PRU. 1PE will expect to have the PRU (to be recovered) in the proper form to be written with a 6681 or a 6684 that is used as a 6681. 1WI

* On coded files (even parity) the tape controller maps an external BCD character of 12 into an internal BCD 00 when the 12 characters is read back. The internal BCD 00 character will be converted into a display code 33. Hence, for a level 12 on coded files, the read driver must search for a display code character 33.

supplies the fwa of the PRU in D.SV1 and the PP byte count in D.SV2. When 1PE has successfully re-written the PRU it will load 1WI back in "on top" of itself and supply the following direct cell communications. Direct cell D.SV1 will be zeroed if the PRU recovered by 1PE was a zero-length PRU of level 17. D.SV2 will contain the byte count of the recovered PRU, and D.FR6-7 will contain the converter and equipment status.

1WI will be entered from 1PE at location 1002B and will then examine the direct cells supplied by 1PE to determine what action should be taken in order to complete the request.

SYSTEM IMPACT

If a sufficient data level is maintained in the circular buffer (two or more PRUs), the driver is able to sustain high speed writing for long periods of time. For properly constructed media conversion programs it would be possible, for example, to write an entire reel of tape without the PP dropping out and releasing the channel. This type of uninterrupted writing is essential if high speed data transfer is to be realized on magnetic tape devices.

Consequently, 1WI will retain the channel without pausing, as long as it is able to maintain high speed tape writing.

END-OF-TAPE

When the end of tape reflective spot is encountered, the end of reel status (2000 octal) is set in the code and status. End of reel processing is determined by the user processing bit, bit 45 in word 2 of the FET.

A. If the UP bit is not set the tape is backspaced over the reflective spot and neither the OUT pointer nor the PRU count is updated. 1WI requests close reel unload by calling CLO in another PP. 1WI stays in its PP and waits for the end of reel flag to be cleared from the FNT indicating that the next reel is ready to be used. At this point 1WI reloads itself to complete the original request on the new reel.

B. If the UP bit is set the PRU count and the OUT pointer are updated and control is returned to the user program.

END-OF-LINE TERMINATOR

During coded writes 1WI must scan each low order byte of a CM word to de-determine if it is a line terminator (a zero byte) and convert it so that it will appear as a 1632 in external BCD. This also applies to coded writes using a 6684. Therefore, a loop to check every fifth byte for a line terminator must be executed when a 6684 is in use. This loop restricts the effective usage of the 6684 hardware in that the data must be "prepared" before writing. This "preparation" time is around 1.7 ms for a 128 word PRU. If, at any time in the future, the BCD PRU size were to be increased then the "preparation" time would also increase and cancel the ability to write tape non-stop on coded writes with a 6684.

17.7 1WS-TAPE WRITE DRIVER

General Description

This driver writes one or more records (a logical record is equal to a physical record) from a circular buffer in central memory to 1/2-inch magnetic tape. No special meaning is attached to any character or series of characters so that the data is written on tape as it was found in the circular buffer. For coded files, of course, converison from display code to external BCD is performed. The only data structure limitations are hardware imposed and consist of limiting a record to containing a multiple of two characters (12 bits) and a maximum length of 5120 characters. The maximum record length constraint applies to both binary and coded records and arises as a consequence of PP buffer size limitations.

1WS recognizes the following functions: WRITEN, WRITE, WRITER, WPHR and WRITEF. The action taken by 1WS for each of the above functions is discussed immediately following the section on principles of operation.

Principles of Operation

The following stratagems have been employed in an attempt to maximize throughput for this type of tape file:

1. Optimize utilization of interrecord gap time,
2. Minimize BCD conversion time, and
3. Optimize use of new hardware features.

Optimum utilization of interrecord gap time is attempted on both binary and coded files. This is exemplified by the steps taken when 1WS is executing in its high speed routine (entered only when a WRITEN function is issued and the circular buffer contains two or more records). The following procedure is a simplified description of the high speed loop.

Step 1 Read the first record from circular buffer into the PP buffer.

Step 2 If binary request (or coded request using a 6684) go to Step 4.

Step 3 If record size is less than or equal to 34 CM words, convert the entire record. If record size is greater than 34 CM words, convert all but the last 72_{10} bytes now and set SW10 to complete conversion during tape start-up time.

Step 4 Begin tape motion (or continue non-stop motion).

Step 5 If binary (or BCD using a 6684) go to Step 7.

Step 6 If SW10 is set to complete conversion, convert the low order 77_{10} bytes of the record.

Step 7 Write record to tape.

Step 8 Read next record from circular buffer into PP buffer.

Step 9 If binary (or BCD using a 6684) go to Step 11.

Step 10 If record size is less than or equal to 34 CM words, convert the entire record and set SW10 to indicate no more converion necessary. If the record size is grater than 34 words convert all but the last 72_{10} bytes now and set SW10 to complete conversion during tape start-up time.

Step 11 Check status on previous record. If parity error call in 1PE to perform recovery procedures. If end-of-tape perform end-of-tape procedures. If status is OK, go to Step 4.

The above algorithm is an example of double buffering, i.e. Record N+1 is read up from CM and converted (if appropriate) during the time that the hardware requires to return end-of-operation on record N, the previous record. Consequently if the user program can keep at least two records in its circular buffer, 1WS will have the next record in PP memory when status is available from writing the previous record. Under the above conditions tape writing will continue in a non-stop manner.

There are three factors that have to be considered in deciding whether or not tape writing in a non-stop fashion can be achieved. They are:

1. The elapsed time from the OAM instruction to the point at which the tape begins to slow down.
2. The time required to read a record from central memory.
3. The overhead contributed by 1WS.

The elapsed time from OAM to slow down (2.6 ms) is a hardware characteristic not under software control. It does, however, indicate the time limit that must be observed for tape writing to be non-stop. Item (2), the time required to bring a record into the PP, is dependent on central memory conflicts (an ambiguity) and the length of the record. Item (3), driver overhead, is a variable but it can be computed and, in fact, is currently in the neighborhood of 0.4 ms/record in the high speed routine. It is apparent, therefore, that only two of the factors governing tape speed are subject to control; namely, driver overhead and record length.

The following examples illustrate the relationship between record size and non-stop writing. For the sake of simplicity it is assumed that no CM conflicts are encountered and that the request is a WRITEN function on either a binary file or on a coded file using a 6684.

For a record size of 400 CM words we have the following timing considerations:

Central Memory read time	2.00 ms/record
1WS inherent overhead	0.40 ms/record
TOTAL . .	2.40 ms/record

The total time to process a record is less than the 2.6 ms limit so that writing will be non-stop.

For a record size of 512 CM words the timings become:

Central Memory read time	2.56 ms/record
1WS inherent overhead	0.40 ms/record
TOTAL	2.96 ms/record

Since this total time exceeds the 2.6 ms limit, the tape will begin to slow down inbetween records.

On a BCD request to write 512 word records, this time assuming NO 6684, there is a radical change in the timing picture caused by the addition of software convert time. The timings are:

Central Memory read time	2.56 ms/record
1WS inherent overhead	0.40 ms/record
1WS software conversion	66.60 ms/record
TOTAL	69.56 ms/record

By taking nearly 70 ms/record it is obvious that the tape will come to a dead stop between records.

Conclusions drawn from the above information:

1. For binary and BCD/6684 files.
   The effective usage of interrecord gap time and the minimization of driver overhead are imperative to maintaining high throughput rates. It is possible to cause a pronounced deterioration in throughput by a very modest increase in driver overhead.

2. For BCD/6681 files.
   The most significant factor is the amount of time necessary to perform the display code to internal BCD conversion within the PP. In fact, interrecord gap time and driver overhead amount to less than five percent of the time required to process a large BCD record. For very small records, however, these considerations play an increasingly major role.

As a result of the importance of BCD convert time (when no 6684 is available) efforts have been made to speed up conversion as much as possible. Two conversion tables are employed and whenever possible interrecord gap time is used to perform part of the necessary conversion.

Full use of the 6684 is made easy by virtue of the tape format. Since no part of the data has any special meaning assigned to it by the operating system (CF. line terminators in SCOPE Standard Tapes) the data does not have to be "prepared" for use with a 6684.

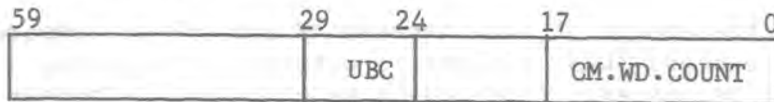
The low speed routine of 1WS may be entered under the following conditions. If the user program is placing data in its circular buffer at a rate approximately equal to the rate at which 1WS is removing data from the buffer and if, at any given instant, 1WS detects only one record remaining in the circular buffer, then 1WS is constrained to operate from its low speed routine. Operating from the low speed routine does not couse the driver to drop out of the PP between records but it does negate effective usage of interrecord gap time. 1WS will return to the high speed routine if, at any given instant, it detects two or more records in the circular buffer. Of course, operating from the low speed routine like the high speed routine, is meaningful only for the WRITEN request since, for any other request, only one data record is processed per PP load (or request).

WRITEN REQUEST

The WRITEN request is the only means under this tape format by which the central processor program can dynamically transfer data from the circular buffer (by placing a header word and data into the circular buffer and advancing the IN pointer while 1WS simultaneously removes data, writes it to

tape and updates the OUT pointer). Thus, the WRITEN request is the only vehicle for sustained high speed writing under this tape format.

When the WRITEN request is used, each data record placed in the circular buffer must be preceded by a control word called the header word. The header word effectively acts as a queue entry for its record and has the following appearance:



where UBC contains the unused bit count of the last word of the record and bits 0-17 contain the central memory word count of the record. The above convention facilitates the writing of records that are a multiple of two characters (12 bits). An error condition is considered to exist (and device capacity exceeded status returned) if UBC is greater than 59, if the CM word count is greater than 512, or if the record size (in PP bytes) is less than or equal to that specified by the installation parameter IP.NOISE, i.e., a noise record.

Writing will continue in the high speed routine if there are at least two records (and their associated header words) in the circular buffer at all times. Writing will continue in the low speed routine for as long as only one record (and its header word) is kept in the circular buffer. Writing from both the high and low speed routines will be interrupted by a parity error, but if the error is successfully recovered, writing will automatically be re-initiated. A similar situation exists at end-of-reel if the UP bit is not set. That is, writing (from high or low speed routines) will be interrupted in order to swap reels. When the new reel is ready to be used, writing will be re-initiated. The FET is not marked complete as a result of either a recovered parity error or an automatic reel swap.

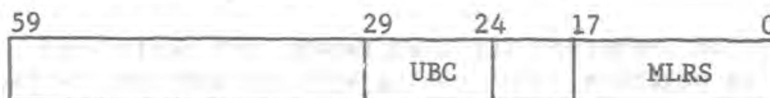
Writing from either the high or low speed routine will unconditionally cease upon detection of an invalid header word or an empty circular buffer.

Both the PRU count and the OUT pointer will be updated after the writing of each record.

The mode for a WRITEN request (and for all requests on an S-tape) is determined by bit 1 of the request code.

WRITE REQUEST

The WRITE request causes one data record to be written from the circular buffer to magnetic tape subject to conditions of word 7 of the FET. Word seven has the following form:



where UBC contains the unused bit count of the last word of the record, and MLRS, the Maximum Logical Record Size field, indicates the maximum number of words that a record may contain.

As with the WRITEN header word, the maximum value of the UBC field is 59 and the maximum value of the MLRS field is 512. If either of these values are exceeded, no data will be written and device capacity exceeded will be returned in the code and status field of the FET. If UBC = 0 and MLRS = 0 SCOPE will assume that the UBC is zero (the last word of the record is a full CM word) and that MLRS = 512. Thus, the default condition (word 7 of FET all zero) permits writing the largest possible records for the device.

When a WRITE is issued the amount of data between IN and OUT is compared to MLRS. If this data exceeds MLRS, nothing is written and a device capacity exceeded error is posted. If the data in the circular buffer is less than or equal to MLRS, this data is adjusted by the unused bit count (UBC) and then written to tape (provided that the byte count is greater than IP.NOISE - cf. WRITEN REQUEST). The PRU count is updated along with the OUT pointer. 1WS then completes the FET and drops out of the PP.

WRITER REQUEST

The WRITER request is identical to the WRITE request.

WPHR REQUEST

The WPHR request causes one data record to be written to tape under the following conditions. If the data between IN and OUT is less than or equal to the value specified by the MLRS field of the FET, the data is adjusted by the unused bit count (UBC field of FET) and the record is written to tape, provided that the adjusted byte count is greater than the declared noise record size (IP.NOISE). The OUT pointer will be incremented to indicate an empty buffer. If the record is a noise record, nothing is written to tape and the status "device capacity exceeded" is returned to the FET. If the data between IN and OUT exceeds the value of MLRS, then device capacity exceeded status is returned, and MLRS words of the data will be written to tape. The OUT pointer will be updated to show that MLRS words of the data have been written.

On a coded WPHR request only conversion from internal to external BCD is performed.

WRITEF REQUEST

The WRITEF request causes an end of file mark (tape mark) to be written on tape. If there is any data present in the circular buffer it will be written to tape first, followed by the file mark. The writing of data from the circular buffer is governed by word 7 of the FET (in a manner identical to the WRITE request).

LOAD POINT

In order to avoid writing on unexpired labelled tapes, 1WS calls OPEN REEL when the PRU count is negative. No writing will be done on the tape until the PRU count goes to zero, indicating that writing is permitted.

PARITY ERROR RECOVERY

By the time the status for a given record has been checked (while operating in the high speed routine), the next reocrd to be written has already been

read into the PP buffer. Hence, if a parity error is encountered, the record sustaining the error must be re-read from CM into the PP buffer. If the record in error is a coded record, it is converted regardless of whether or not a 6684 is available (this is because 1PE does not use a 6684 during recovery procedures).

At this point 1WS calls the write parity error recovery routine (1PE) in "on top" of itself as soon as it has set up the following direct cell communications:

CELL	CONTENTS
D.SV1	FWA OF DATA IN PP BUFFER
D.SV2	DATA BYTE COUNT

After 1PE has successfully recovered the parity error it sets up the following direct cell communications for 1WS.

CELL	CONTENTS
D.FR6	CONVERTER STATUS
D.FR7	EQUIPMENT STATUS

1PE calls in 1WS (into the same PP) and cuases 1WS to receive control at location 1002B.

In order to complete the request 1WS must examine the unit status supplied by 1PE for end-of-tape, since 1PE could have encountered end-of-tape in attempting to recover the parity error.

If end-of-tape (reel) was detected 1WS will perform end-of-reel procedures, and, based on the UP bit, complete processing of the request. If end-of-tape was not encountered, the request will either be completed, or, in the case of a WRITEN request, it will be resumed.

SYSTEM IMPACT

If a sufficient data level is maintained in the circular buffer (two or more records), the driver is able to sustain high speed writing for extended periods of time, e.g., an entire reel of tape. This type of uninterrupted writing is essential if high speed data transfer is to be realized on magnetic tape devices. Consequently, 1WS will retain the channel as long as it is able to write data from the circular buffer.

END-OF-TAPE

When the end-of-tape reflective spot is sensed, the end of reel status (2000 octal) is set in the code and status. End-of-reel processing is determined by the user processing bit, bit 45 of word 2 of the FET.

A. If the UP bit is not set, 1WS requests close reel unload by calling CLO in another PP. 1WS stays in its PP (the channel is dropped, of course, and the equipment has been released) and waits for the end-of-reel flag to be cleared from the FNT, indicating that the next reel is ready for use. At this point 1WS reloads itself to complete the original request on the new reel

B. If the UP bit is set, the PRU count and OUT pointer are updated, the equipment is released and control is returned to the user program.

TIMING MEASUREMENTS

Through-put of the 1WS write driver was measured and recorded on the chart below. The measurements were under "ideal" conditions, that is, only the test program was in execution and it was coded so that it could make maximum use of the WRITEN request.

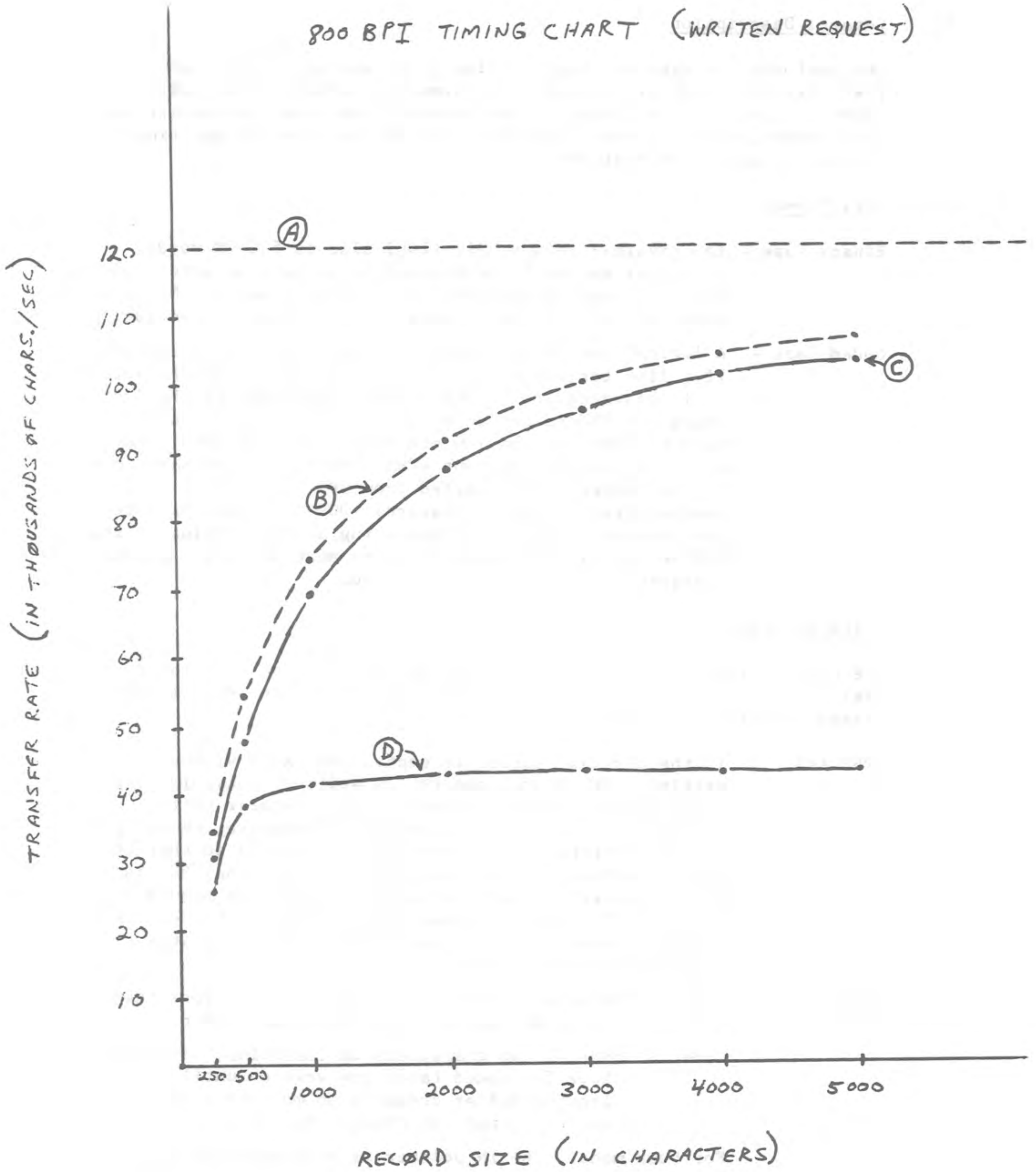
The four curves, A through D, have the following significance:

Curve A. This curve represents the transcribing rate of a 607 tape transport writing at 800 BPI.

Curve B. This curve represents the theoretical maximum transfer rate as a function of record size and approaches curve A asymptotically with increasing record size.

Curve C. This curve represents the observed through-put rate as a function of record size (for the sizes shown at the bottom of the chart along the horizontal axis) for either a binary request or a coded request that is executed using a 6684.

Curve D. This curve reflects the observed through-put rate as a function of record size (for the sizes shown at the bottom of the chart along the horizontal axis) for a coded request executed on a 6681 converter.


800 BPI TIMING CHART (WRITEN REQUEST)
TRANSFER RATE (IN THOUSANDS OF CHARS./SEC)
120
110
100
90
80
70
60
50
40
30
20
10
A
B
C
D
250
500
1000
2000
3000
4000
5000
RECORD SIZE (IN CHARACTERS)

17.8 1WX - WRITE EXTERNAL TAPES

I. General Description

1WX performs the external tape writing which was previously done by 2TW. External tape processing is retained for compatibility with SCOPE 2.0 and earlier systems. The external tape write procedure has been redesigned for greater efficiency using inter-record gap time to prepare to write the next PRU.

II. Data Format

Binary Tape - The physical record unit (PRU) size is 512 CM words. A logical record is terminated by a short or zero length PRU. The zero length PRU is 4 bytes of zeros. No level numbers are appended to external tape logical records.

Coded Tape - A logical record is a physical record of 136 characters. If a line terminator is encountered before 136 characters, it is converted to blanks and the remainder of the 136 character PRU is set to blanks. A line terminator is 12 bits of zeros in the low order byte of a CM word. For all write requests except WPHR, the characters taken from the CM buffer are converted from DPC to internal BCD, then written on tape in external BCD. If 136 characters are converted without encountering a line terminator the OUT pointer is incremented to the next CM word, so that characters 137-140 are not written.

III. Write Requests

The type of request and mode are determined by the code and status field in word 2 of the FST entry. There are four write requests processed by 1WX:

WPHR (4) If the circular buffer is empty, 1WX exits without writing. Otherwise, one PRU is written containing 512 words or the number of words in the circular buffer; whichever is less. For a coded WPHR request, the display code to Internal BCD conversion is bypassed so that the PRU is converted from display code to external BCD as it is written on tape. If a WPHR request is issued with more than 512 words in the circular buffer, the device capacity exceeded bit is set in the code and status of word 2 of the FST entry.

WRITE (14) Binary - PRUs of 512 words are written until less than 512 words remain in the circular buffer.

Coded - PRUs of 136 characters in the format described above for coded tapes are written until the circular buffer contains no more line terminators and less than 136 characters.

For both modes, the IN pointer is read prior to each block read from central memory so that 1WX will write continuously as long as the user keeps enough data in the buffer.

WRITER (24) The IN pointer is not reread by 1WX so that the number of words to be written cannot be altered after the request is issued.

Binary - 512 word PRUs are written until less than 512 words remain in the circular buffer. If any words are remaining, they are written as a short PRU. Otherwise, a zero length PRU is written. The short or zero-length PRU indicates end-of-logical record.

Coded - The coded WRITER is the same as the coded WRITE except that if less than 136 characters remain in the buffer with no line terminator, this data is expanded with blank fill to 136 characters and written as another PRU.

WRITEF (34) Buffer not empty:
1WX performs the WRITER function, then writes a file mark.

Buffer empty:
Binary - If the last operation was WRITE, a zero length PRU is written to end the previous logical record, then a file mark is written. If the last operation was not WRITE, only a file mark is written.

Coded - A file mark is written.

IV. Beginning of Information (Negative PRU Count)

If the PRU count is negative and the unit is busy, 1WX re-issues the CIO call with a delay and drops out to avoid waiting in a PP during a REWIND. If the PRU count is negative and the unit is not busy, 1WX calls OPE, then re-issues the CIO call with a delay and drops out.

V. End-of-Reel Processing

When the next PRU is ready to be written, 1WX loops until end-of-operation on the previous PRU, then tests for parity error or end-of-reel status. If end-of-reel status is sensed after 1WX has detected end-of-operation, end-of-reel processing will not be done until the next PRU is written. If no parity error was encountered, 1WX checks the UP bit in word 2 of the FET. If the UP is on, 1WX sets end-of-reel status in the FET and FST code and status fields, updates the FET and FST, and returns to the user. If the UP bit is off, 1WX calls CLOSE, REEL in another PP, then loops pausing and checking the code and status of the FST until end of reel is cleared. The request is completed on the new reel.

VI. Parity Errors

Before the status is checked on the last PRU written, a new PRU is constructed in the PP buffer. If a parity error is detected, that last PRU written is reconstructed and 1PE is called to recover. On return from 1PE, if end-of-reel status is set, the end-of-reel processing is performed before completing the request

VII. General Logic Flow

The WPHR, WRITER, and WRITEF functions are shown in the 1WX flowchart. The WRITE function in binary mode is performed as follows:

1. The first PRU is read from OUT and written on tape. The IN pointer is read from central memory.
2. The next PRU is read from the circular buffer.
3. 1WX waits for end-of-operation on the last PRU written. If no parity error was encountered, the PRU count is updated in PP memory and the FET OUT pointer is updated in central memory. If end-of-tape status was set, end-of-reel processing is performed.
4. The next PRU is written and the IN pointer is re-read from central memory. If there is at least one PRU remaining in the buffer, the write loop continues at step (2) above.

The coded WRITE procedure differs from the binary WRITE function in that more than one PRU can be read from the circular buffer with each central read instruction. For the first central read, the maximum word count used is the number of words that will fit in the PP coded buffer area. PRUs are formatted from data in this PP coded buffer into a 136 character PRU buffer, so that a central read is issued only when the PP coded buffer is empty instead of after each PRU written as in the binary case. In order to keep the tapes moving across record gaps, the maximum word used for subsequent central read instructions is set to 30. The IN pointer is read from central memory before each central read.

VIII. Entry Information

The following low core cells must be set on entry to 1WX:

D.FNT (20-31)	Words 1, 2 of FST entry
D.EST (32-36)	EST entry
BS (45)	Last buffer status
D.PPIRB (50-54)	Input Register
D.RA (55)	Relative Address
D.FL (56)	Field Length
D.FA (57)	FST address
D.FIRST (60-61)	
D.IN (62-63)	
D.OUT (64-65)	
D.LIMIT (66-67)	
D.CPAD (74)	Control Point Address
D.PPIR (75)	Input Register Address
D.PPOR (76)	Output Register Address
D.PPMES1 (77)	Message Buffer Address

In addition, the following cells must be set when 1WX is entered from 1PE:

ST (47)	Unit Status
DL (72)	Number of PP words in the PRU written by 1PE.

IX. Other Routines Called

1. 1PE.
   1PE is called to recover from a parity error. In addition to the entry information listed above, 1WX sets:

DA (71)	PP address of the PRU to be rewritten,
DL (72)	Number of PP words in the PRU located at DA,
CMDL (73)	Number of CM words to update OUT by after the PRU is successfully written.

   1PE is expected to update the PRU count in PP low core and the OUT pointer in central memory, reload 1WX, and jump to 1002 with unit status in ST (47) and the entry information set.

2. OPE
   OPE is called in another PP if the PRU count is negative. The input register for the open reel with rewind function has bit 41 set as an internal call flag and bits 24 and 29 set to indicate that the call came from 1WX. 1WX sets bit 39 of its input register to show that OPE has been called, issues a request peripheral job with this input register and a 2 second delay, and drops from the PP. 1WX continues to recall itself with a delay and drop out each time it is called until the PRU count becomes positive indicating that the OPEN function was completed.

3. CLO
   CLO is called in another PP. The input register for the close reel unload function has bit 41 set as an internal call flag and bits 24 and 29 set to indicate that the call came from 1WX. After calling CLO, 1WX loops waiting for end-of-reel to be cleared in the FNT.

X. 1WX Timings

X = maximum possible transfer rate at density = d, assuming no record gaps.

$$X = \frac{d*\text{tape speed}}{10^3}\ \text{KC}$$

$$X = \begin{cases} 30\text{KC for } 200\text{ BPI} \\ 83.4\text{KC for } 556\text{ BPI} \\ 120\text{KC for } 800\text{ BPI} \end{cases}$$

Y = % of this which contains data, assuming PRU size = P, density = D, record gap = 3/4 inch.

$$Y = \frac{P/d}{\frac{P+n*4}{d} + \frac{3}{4}} \qquad \text{where } n = \begin{cases} 1 \text{ for } 200\text{ BPI} \\ 1 \text{ for } 556\text{ BPI} \\ 2 \text{ for } 800\text{ BPI} \end{cases}$$

Z = X*Y = theoretical maximum transfer rate at density d, PRU size P, with 3/4 inch record gap.

	BCD			BINARY		
	200 BPI	556 BPI	800 BPI	200 BPI	556 BPI	800 BPI
theoretical	14.07	20.40	21.94	29.12	77.1	107
700 ft. * Unit "E"	9.27	16.32	18.07	29.17	75.45	103.46
in/RPU	1.47	1.00	0.93	26.37	9.966	7.16
PRU/700ft.	5794	8400	9033	319	840	1173

* 700 ft. is an approximate length

PP LOW CORE CELL ALLOCATION FOR 1WX

ADR	SYMBOL	VALUE
1		
2		
3		
4		
5	PRUIN	IN POINTER FOR PRUBF
6	OUTPP	CM OUT POINTER FOR CURRENT PRU
7	↓	
10	CM	TEMPORARY STORAGE USED FOR CRD
11		
12		
13		
14	↓	
15	CH	CHANNEL NUMBER
16	PPNEXT	OUT POINTER FOR BCD PP BUFFER
17	N	WORD COUNT FOR NEXT CRM
20	D.FNT+0	FNT WORD 2
21	+1	
22	+2	
23	+3	
24	↓ +4	

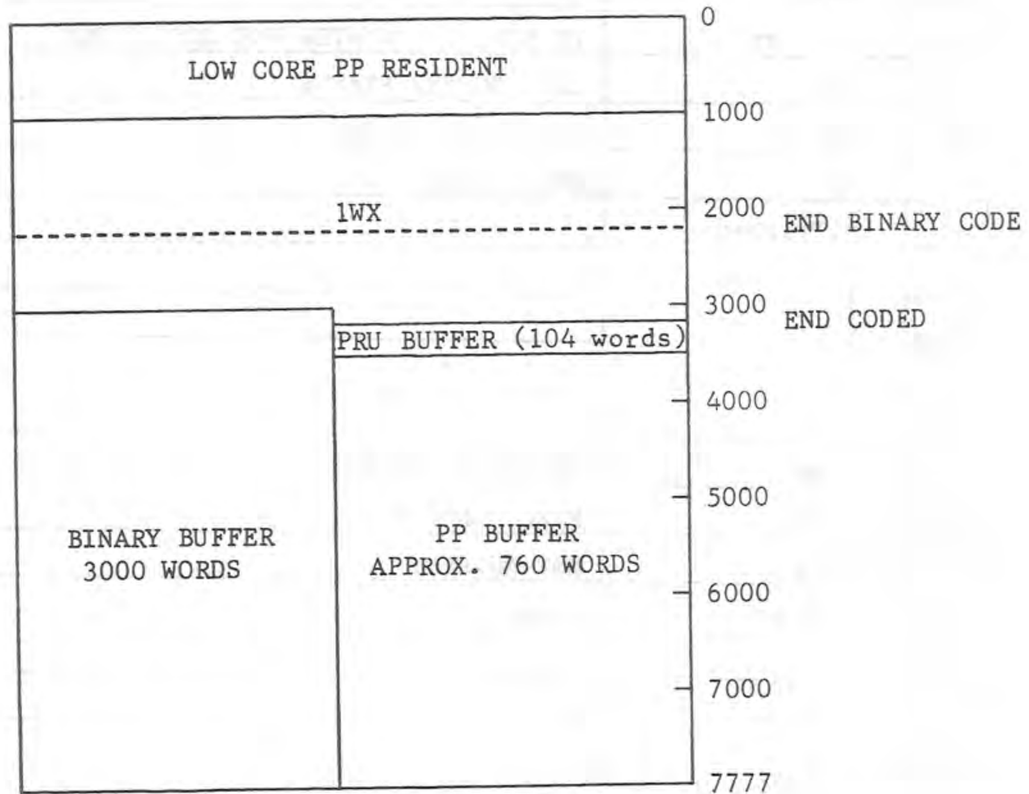
ADR	SYMBOL	VALUE
25	D.FNT+5	FNT WORD 3
26	↓ +6	
27	+7	
30	+8	
31	↓ +9	
32	D.EST+0	EST ENTRY
33	↓ +1	
34	+2	
35	+3	
36	↓ +4	
37	FN	CURRENT FUNCTION CODE
40	K	NUMBER OF CM WORDS TO WRITE IN NEXT PRU
41	PPOUT	CM FWA OF NEXT PRU
42	↓	
43	PPIN *	CM LWA+1 OF NEXT PRU
44	↓ **	IF BIN, IN POINTER FOR BUF IF BCD
45	BS	LAST BUFFER STATUS
46	CS	CONVERTER STATUS
47	ST	UNIT STATUS
50	D.PPIRB+0	
51	↓ +1	
52	+2	
53	+3	
54	↓ +4	
55	D.RA	RELATIVE ADDRESS
56	D.FL	FIELD LENGTH
57	D.FA	FNT ADDRESS
60	D.FIRST	FIRST
61	↓	
62	D.IN	IN
63	↓	
64	D.OUT	OUT
65	↓	

* 43 also used for WTG = words to go if binary
** 44 also used for BTG = blocks to go if binary

ADR	SYMBOL	VALUE
66	D.LIMIT	LIMIT
67	↓	
70		
71	DA	DATA ADDRESS OF PRU TO BE REWRITTEN IF PAR. ERR.
72	DL	DATA LENGTH=NUMBER OFF PP WORDS STARTING AT DA
73	CMDL ***	NUMBER OF CM WORDS TO UPDATE OUT BY IF PAR. ERR.
74	D.CPAD	CONTROL POINT ADDRESS
75	D.PPIR	INPUT REGISTER ADDRESS
76	D.PPOR	OUTPUT REGISTER ADDRESS
77	D.PPMES1	PP MESSAGE BUFFER ADDRESS

*** 73 also used for KL = size in CM words of last PRU written if binary

1WX MEMORY USAGE



17.9 2TB

Function

2TB performs all backward positioning of one half inch magnetic tape.

Characteristics

The functions handled by 2TB:

BACKSPACE	40
BACKSPACE PRU	44
REWIND	50
UNLOAD	60
SKIP BACKWARD	640

Before positioning backwards, 2TB checks the last buffer status. If the last operation was a write or an open write, the trailer label is written before the tape is moved.

First a tape mark is written. If the requested function is modified by 300 (for example, 340 for backspace or 350 for rewind) an EOV1 label is to be written. Otherwise, an EOF1 label is to be written. 4LB is called to write the appropriate label and two more tape marks are written. If a zero function code is requested the tape is positioned before the first tape mark if an unlabelled file and before the last tape mark if labelled.

Rewind, unload and BKSPRU are accomplished by issuing the appropriate hardware functions.

Backspace positions backward to the last logical record by reading backward to the first short PRU and positioning after it.

Skip backward performs like backspace with the option of positioning after a record of a specific level and iterating to this level the number of times specified by the N parameter. The level number is upper four bits of the function code and the N parameter is in bits 18-35 of the input register.

17.10 1TF

Introduction

1TF is loaded by CIO to process SKIPF requests (240_8) on magnetic tape files. Such a request may have been issued by the user or may have been generated by 1RT for terminal processing of a READSKP request (20_8). In either case, 1TF will finish the request and set completion bits in the FNT and FET.

Entry Information

1TF expects PP low core to be set up as follows:

Symbolic Address	Content	Actual Address
D.FNT - D.FNT+4	FNT (2)	20-24
D.FNT+5 - D.FNT+9	FNT (3)	25-31
D.EST - D.EST+4	EST	32-36
D.BA - D.BA+4	FET (1)	40-44
D.PPIRB - D.PPIRB+4	Input register	50-54

The skip count is in bits 18-35 of the input register. The level number is in bits 14-17 of FNT (3).

Initiallization

The tape unit status is checked for busy. If the unit is ready, but busy, 1TF will recall CIO with a 2 second delay. If the PRU count is negative, 1TF calls OPE. Otherwise, main loop initiallization is done. Noise record size is set to IP.NOISE unless the file is in standard SCOPE format in which case, noise size is unconditionally set to 6 characters. If the skip count is 777777_8, the function is a no-operation, and exit processing will be performed. Otherwise, a table loop-up is performed to locate the address of the control routine used to perform the skipping as follows:

SCPD	Skip 1 logical record on standard SCOPE formatted coded file.
SCPBIN	Skip 1 logical record on standard SCOPE formatted binary file.
SLCD	Skip 1 physical record on coded S- or L- formatted file.
SLBIN	Skip 1 physical record on binary S- or L- formatted file.

XBIN	Skip 1 logical record on binary X- formatted file
XCODED	Skip 1 physical record on coded X- formatted file.

Unless the file is in standard SCOPE format, any level number except 17_8 will be mapped into 0.

Main Loop Processing

The skip routine selected is called to skip one record until the specified number of records of the specified level have been read. The skip count will be decremented by one for each record with a level number greater than or equal to the specified level number.

Exit Processing

When the skip count is satisfied, the end-of-record bit and the last level number read are set into the FET and FNT code and status fields. If the last level read was 17_8, the end-of-file bit will also be set. The FET and FNT completion bits are set and the PP is dropped.

End-of- reel Processing

The CALLRP subroutine will be called if any of the following conditions occur during a record read:

SLCD/SLBIN XBIN/XCODED	If the reflective spot is encountered; if a tape mark is read and the file is labelled. In the latter case, the tape mark may or may not precede a label. X-tapes cannot be labelled.

CALLRP sets the end-of-record bit, the last level number read, and, if appropriate, the end-of-file bit into the FET and FNT. A check is made to see if the skip count was satisfied by the last read; if so, the skip count in the input register is set to zero. The end-of-reel and/or end-of-information bits are set into the FNT if the reflective spot and/or a tape mark, respectively, were read. The end-of-file and end-of-record bits are then cleared in the FNT; 3RP will reset them before dropping out. The PRU count is decreased by one; 3RP will re-increment it if the record was not part of a label group. 3RP is then loaded.

NOTE: If 3RP determines that a reel swap is necessary, the SKIPF or READSKP request will be considered to be complete, whether or not the skip count has been satisfied. While this vastly reduces the utility of the SKIPF/READSKP request, it is considered preferable to inconsistencies that would otherwise arise. (There is no way to communicate to the user program the number of records left unskipped; this means that a program would operate on a particular tape file differently, depending on whether or not the UP bit is set). End-of-reel processing for SCOPE-formatted tapes is performed by calling CLO into another PP.

Messages

MT uu REJECT

appears if the unit rejects a function; 1TF loops until the condition is corrected or the job is dropped.

MT uu XMSN PARITY ERROR

appears if a transmission parity error occurs when a function is issued; a master clear is performed and 1TF loops until the condition is corrected or the job is dropped.

MT uu NOT READY

appears if the unit is not ready; 1TF loops until the unit is ready or the job is dropped.

MT uu RESERVED

appears if the unit is reserved; 1TF loops until the unit is free or the job is dropped.

MT uu BLANK TAPE READ

appears if no data is received from the channel after 1 second; exit procedures are performed.

17.11 4LB

4LB is called to process the standard SCOPE 3.0 1/2 inch magnetic tape labels. Processing may take several forms which are broadly classified below as READ Section, WRITE Section and Tape Motion Section of 4LB.

The data from which the labels are formed is taken from the FNT, the FET and the Control Point area. The FNT contributes the block count (PRU number) and the density setting. The last four words of the FET provide all other user controlled label information. These fields are used directly, when given and not in format error, except for the retention cycle which is added to the creation date to produce an expiration date for the label. The Visual Reel Number is transferred to 4LB via the Control Point Area. 4LB will pause and request this number from the operator whenever it is needed.

The FNT and FET also provide the data against which the label is checked when that operation is requested. If an expiration only check is requested, the expiration date from the label is compared with today's date contained in Central Memory. When a label is delivered to Central Memory, 4LB does not update the IN pointed but leaves that operation to the calling routine.

HDR, EOF and EOV Labels

Field	Value
Label Identifier	Input in bits 0-1 of the Communication word
File Label Name	Either the FET entry or Blanks are used
Multi-file Identification	Either the FET entry or Blanks are used
Reel Number	Either the FET entry or 0001 is used. If the FET entry is blank, 0001 is inserted in the FET
Multi-file position Number	Either the FET entry or blanks are used
Edition Number	Either the FET entry or 01 is used. If 01 is Used, the edition number In the FET is set to 01
Creation Date	Either the current date Obtained from SCOPE (which is inserted in the FET) or word 12 of the FET is used
Expiration Date	1) Creation date or 2) Calculated by adding the

	Creation date to the Retention cycle module 366(10)
Block Count	HDR1- set to zeros EOV1 PRU count of the FNT EOF1 PRU count of the FNT

VOL Labels

Visual reel number	From Control Point Area
Density	FNT entry word 2

Messages

All messages are preceeded by MT XX LLLL where LLLL is the label ID

A. FILENAME SHOULD BE XXX-XXX IS XXX-XXX
B. FILE NAME READ XXX-XXX.
C. REEL Number should be XXXX IS XXXX.
F. CREATION DATE SHOULD BEXXXXX IS XXXXX.
G. BLOCK COUNT SHOULD BE XXX IS XXX
H. EXPIRATION DATE SHOULD BE XXXXX- IS XXXXX.
I. MULTI-FILE NAME SHOULD BE XXX IS XXX.
J. FILENAME WRITTEN XXX-XXX.
K. VISUAL REEL NUMBER WRITTEN XXX-XXX.
L. VISUAL REEL NUMBER READ XXX-XXX
M. ENTER VISUAL REEL NO.
N. REJECT.
O. XMSN PAR ERR
P. NOT READY
Q. RESERVED
R. NO WRITE ENABLE
S. LABEL INFO ERR IN FET.
T. LABEL PARITY ERR
U. BLANK TAPE READ

Entry Information

Before calling 4LB a communication word--CW--is set to low core to direct 4LB as to what action should be performed. The following settings of CW are meaningful:

Read section of 4LB

(CW) = 10B-- read and check record
(CW) = 11B-- skip forward to tape mark (done by reading a physical record, pausing and checking for a tape mark)
(CW) = 14B-- read and do not check record
(CW) = 15B-- read and check expiration only

```
    {CW} = 16B-- read, check and, if HDR, deliver to CM
    {CW} = 17B-- read tape mark
    {CW} 12B, 13B undefined

Write section of 4LB
    {CW} = 0 -- write EOV label
    {CW} = 1 -- write EOF label
    {CW} = 2 -- write VOL label
    {CW} = 3 -- write HDR label
    {CW} = 7 -- write tape mark
    {CW} = 4, 5, 6 undefined

Tape motion section of 4LB
    {CW} = 30b -- rewind tape
    {CW} = 31B -- rewind unload tape
    {CW} = 32B -- backspace 1 physical record
    {CW} = 33B -- skip forward tape mark
    {CW} = 34B -- skip backward tape mark
    {CW} = 35B -- write tape mark*****do not use, may hang
                  PP
    {CW} = 36B -- skip bad spot
    {CW} = 37B -- set density {according to what is in
                  D.FNT}
```

Exit Information

```
CW will be set to indicate the status of the data which
was read as follows:
    {CW} bits 0-2
    {CW} = 0 -- EOV read
    {CW} = 1 -- EOF read
    {CW} = 2 -- VOL read
    {CW} = 3 -- HDR read
    {CW} = 4 -- record read was not recognized
    {CW} = 7 -- tape mark read
    {CW} = 5, 6, undefined
    {CW} bit 3 -- on indicates multi-file name error
    {CW} bits 4-5 these bits set only if FET MFN is
    Non-zero
         = 00 -- label multi-file pos. No. Eq FET multi-
                 file Pos. No.
         = 01 -- label multi-file pos. No. LT FET multi-
                 file Pos. No.
         = 10 -- label multi-file pos. No. GR fet multi-
                 file No.
    {CW} bits 6-7
         = 00 -- label reel number EQ FET reel number
         = 01 -- label reel number LT FET reel number
         = 10 -- label reel number GR FET reel number
    {CW} bit 8 -- on indicates file label name error
    {CW} bit 9 -- on indicates tape not expired
    {CW} bit 10 -- on indicates block count does not agree
```

{CW} bit 11 -- on some other field in error

If HDR label was read or written D.SV1 and D.SV2 contain the multi-file position number

{CW} is meaningless when the call was for Write or Tape motion except that if {CW} = 3 on entrance to 4LB and the tape is at load point, {CW} will be set equal to 2 and no action will be performed}

17.12 4LC

4LC is the PP routine, the Y type processor, that processes standard 3000 series labels. The features and restrictions are:

A. 4LC will rewind and unload files as directed by OPE and CLO.

B. The label information is validated against the contents of the FET and any discrepancies are immediately made known to the operator. 4LC will wait for the operator to type GO, RECHECK or DROP.

C. Multi-reel processing is initiated from CIO when an end of reel has been detected. CLO is called and 4LC is directed to read or write an EOT label and to rewind or unload the file. CLO then directs further action based on the setting of the UP bit.

D. Multi-file reels are handled simply by issuing repeated CLOSE NO REWIND/OPEN NO REWIND functions.

E. Non-standard labels are handled in the following way:

   1. 4LC will not read or write the header, the user must take care of this. 4LC will perform the motion requested by OPE and that is all.

   2. Trailer label processing differs for input and output tapes. For output reels the standard EOF or EOT labels are written. For input reels an invalid trailer message is given to the operator and if his response is GO, normal processing continues.

F. The following items are validated agains the FET if they are supplied for an input tape:

   a. label file name
   b. reel number
   c. creation date
   d. edition number

G. For an output tape the expiration date is checked to see if the tape has passed thru its retention cycle.

H. Blank labeling capability is provided by the simple scheme of 4LC posting a message to the operator when an output reel does not contain a valid header. The message "NOT Y TAPE" appears and a response of GO causes the tape to be rewound and the header written. We can not handle a virgin tape.

I. The header label record is not delivered to the circular buffer only to the FET.

J. The block count {PRU COUNT} in the EOF or EOT labels does not include label records or tape marks.

K. The 48 character user area in the header label is always blank since there is no FET address as yet that directs 4LC to the area.

Entry Information

Before calling 4LB, a communications word --CW-- is set in low core to direct 4LC as to what action should be performed. The following settings of CW are meaningful:

Read Section of 4LC

```
{CW}=10B--READ AND CHECK RECORD
{CW}=15B--CLOSE REWIND CALL FROM *CLO* TRLR ALSO READ
{CW}=16B--CLOSE UNLOAD CALL FROM *CLO* TRLR ALSO READ
{CW}=17B--SEARCH FOR EOF/READ TRLR/SKIP EOF**CALLED
          FROM CLO**
```

For Open Rewind calls 1MR sets 2 to the fifth of CW.

Write Section of 4LC

```
{CW}= 0 --WRITE EOT LABEL
{CW}= 1 --WRITE EOF LABEL
{CW}= 3 --WRITE HDR LABEL
{CW}= 4 --WRITE EOT/REWIND FROM CLO
{CW}= 5 --WRITE EOF/REWIND FROM CLO
{CW}= 6 --WRITE EOT/UNLOAD FROM CLO
{CW}= 7 --WRITE EOF/UNLOAD FROM CLO
```

Exit Information

CW will be set to indicate the status of the data which was read as follows:

```
{CW}= 0 --EOT READ
{CW}= 1 --EOF READ
{CW}= 2 Not used
{CW}= 3 --HDR READ
```

Error conditions are handled locally in 4LC, giving the operator the chance to **GO**RECHECK**DROP**

Y Tape Operational Guide

The dayfile messages written by 4LC are of an informative type or an action type. The action type messages require an operator response of N.GO, N.RECHECK or N.DROP. All of the messages are preceded by MTXX to identify the unit in question.

MESSAGE	TYPE	REPLY
1 - LBL WRITTEN	Informative	-
2 - LBL READ	Informative	-
3 - BLKERR FNT XXXX	Informative	-
4 - CDATE-NON NUMERICS- TODAYS USED	Informative	-
5 - 4LC ILLEGAL FX XX	Informative	-
6 - NOT Y TAPE	Action	GO/RECHECK/DROP
7 - UNEXPIRED	Action	GO/RECHECK/DROP
8 - LBLNAM ERR	Action	GO/RECHECK/DROP
9 - EDITNO ERR FT XX	Action	GO/RECHECK/DROP
10 - REELNO ERR FT XX	Action	GO/RECHECK/DROP
11 - CDATE ERR FT YYDD	Action	GO/RECHECK/DROP
12 - INVALID HDR	Action	GO/RECHECK/DROP
13 - INVALID TLR	Action	GO/RECHECK/DROP

1 - LBL WRITTEN - This precedes the 80 character header or trailer record message for output tapes.

2 - LBL READ - This preceds the 80 character header or trailer record message for input tapes.

3 - BLKERR FNT XXXXX - Indicates that the block count in the trailer label is not equal to the block count in the FNT.

4 - CDATE-NON NUMERICS - The creation date that is supplied in the FET contains a non-numeric character. The header label is written using today's date as the creation date.

5 - 4LC ILLEGAL FX - 4LC was called with a function code that could not be interpreted. The channel and equipment are released and the control point is aborted.

6 - NOT Y TAPE - Indicates that the output tape being processed does not contain a 3000 series label. A response of N.GO causes the label to be written and processing continues.

7 - UNEXPIRED - Indicates that the output tape being processed has not reached the end of its retention cycle. A response of N.GO causes the tape to be relabeled and processing continues.

8 - <u>LBLNAM ERR</u> - The label name in the FET does not match the label name in the header. A response of <u>N.GO</u> causes the header label name to be accepted as the valid name and processing continues.

9 - <u>EDITNO ERR FT XX</u> - The FET edition number {FT XX} does not match the header edition number. A response of <u>N.GO</u> causes the header edition number to be accepted as the valid one and processing continues.

10 - <u>REELNO ERR FT XX</u> - The FET reel number {FT XX} does not match the header reel number. A response of <u>N.GO</u> causes the header reel number to be accepted as the valid one and processing continues.

11 - <u>CDATE ERR FT YYDDD</u> - The FET creation date {FT YYDDD} does not match the header creation date. A response of <u>N.GO</u> causes the header creation date to be accepted as the valid one and processing continues.

12 - <u>INVALID HDR</u> - The header label read was not recognized as a standard 3000 series label. A response of <u>N.GO</u> causes the error bits to be set in the status field of the FET and error processing continues from these.

13 - <u>INVALID TLR</u> - Indicates that during close file or close reel processing an unrecognizable trailer label was read. The response <u>N.GO</u> causes the trailer to be accepted and the file will be rewound or unloaded as the close function dictates. Normal processing will then continue.

CHAPTER 18 - TABLE OF CONTENTS

18.1	Introduction	18-1
18.2	Functions	18-8
18.3	PFM Main Line	18-12
18.4	Routine Details	18-14
18.5	PFC	18-14
18.6	PFA	18-22
18.7	PFP	18-28
18.8	PFE	18-29
18.9	Common Routines	18-39
18.10	System Interface	18-71
18.11	AUDIT	18-75
18.12	DPF - DUMPF	18-85
18.13	LPF - LOADPF	18-98

Permanent Files Implementation

18.1 Introduction

For Permanent File Phase 1, the regular SCOPE System contains four extra entities. These are called the Permanent File Directory (PFD), the RBT Catalogue (RBTC), the Attached Permanent File Table (APF), and the Sub-Directory Table (SDT).

The PFD and the RBTC reside on mass storage (both on the same device) and are of fixed length. At Dead-Start time, an FNT entry is made for the RBTC as 0RBTC, and multiple entries are made for PFD as 0SD000, 0SD001, 0SD002,....; the 0SD000 being for the PFD header and the others corresponding to each of the Sub-Directories. Both the RBTC and all the PFD entries are attached to control point zero and entered at the high end of the FNT. This prevents an unauthorized user from interfering with the operation of the Permanent File Manager (PFM).

The APF table is CM resident and consists of one word entries, up to a predefined size for the installation, with an overall maximum of 511 entries.

The SDT is also CM resident and consists of a one byte entry per Sub-Directory (SD) other than SD0. The number of SD's may vary from 2 to 1000.

The PFD is detailed in figure (a). As can be seen, it consists of a header followed by a number of sub-directories of equal length. The PFD header is a copy of the PFD's RBT chain for use in recovery. Each sub-directory consists of PFD entries, the format of which is shown in figure (b).

Up to five cycles or versions of each Permanent File (PF) can be maintained. The PFD entry, therefore, can have up to five pointers to the RBTC, each corresponding to an entry for a cycle.

Each pointer has a byte associated with it (word 6 of PFD entry), which will contain two, one bit flags and the cycle number. The first flag, if on, will signify that this cycle has been dumped and is now unavailable, the second will signify the entry is incomplete (i.e., no RBTC entry exists at this time).

When cataloguing (see 3.1) the user must specify the number for the cycle unless it is the first cycle of a new file. When attaching a file, unless the user specifies the cycle he wishes, he will get the one with the highest cycle. Cycle numbers must lie between 1 and 63 inclusive.

The Permanent File Name (PFN) can consist of up to 40 characters. The PFD entry also contains the passwords given when the file was catalogued. Details of the password scheme are given in the write up of CHPERM elsewhere in this IMS.

The 'utilities flag' in word one of the PFD entry will signify that all cycles have been dumped. (Used by utilities only.) The 'entry in use' flag is used when an entry is built. (1=IN USE.)

Figure (c) is the detailed format of the RBTC and figure (d) is that of a RBTC entry.

The RBTC consists of a header followed by a number of variable length RBTC entries. These entries are not necessarily contiguous. If an RBTC entry will not fit entirely into one PRU, it is forced to start at the beginning of the next PRU, i.e., by making an entry start at the start of a PRU, it will be able to cover several consecutive PRU's. The header is a copy of the RBT chain for the catalogue as a whole for use in recovery. Each RBTC entry has a copy of the PFN as well as the other relevant information shown in the figure. This is followed by the RBT chain for the file.

NAW, NAE are reserved for future use. W is the current number of words in entry.

As can be seen in figure (d), two variable length fields have been left within the entry for installation and/or future development use.

Figure (e) is the format of the APF entry. Whenever a permanent file is attached to a control point, an entry is made for it in the APF table. A nine-bit pointer is used to relate each FNT entry to the corresponding APF entry. The APF table is really an interlock list to allow multi-read access without reloading the RBT chains and also controls the queueing for single or priority use of the file.

There are five one bit flags in the APF word:

PRIORITY FLAG	:	Used to signify that either someone is waiting on exclusive access or has got exclusive access.
WAIT FLAG	:	Used to signify that PFM is waiting to use this file at completion of current operation.
PURGE FLAG	:	Is used to prevent any new operations from starting on a file after the file has been logically removed from the system by PFM.

Figure {a}

THE PERMANENT FILE DIRECTORY

{Contiguous Area, Reserved at Dead-Start}

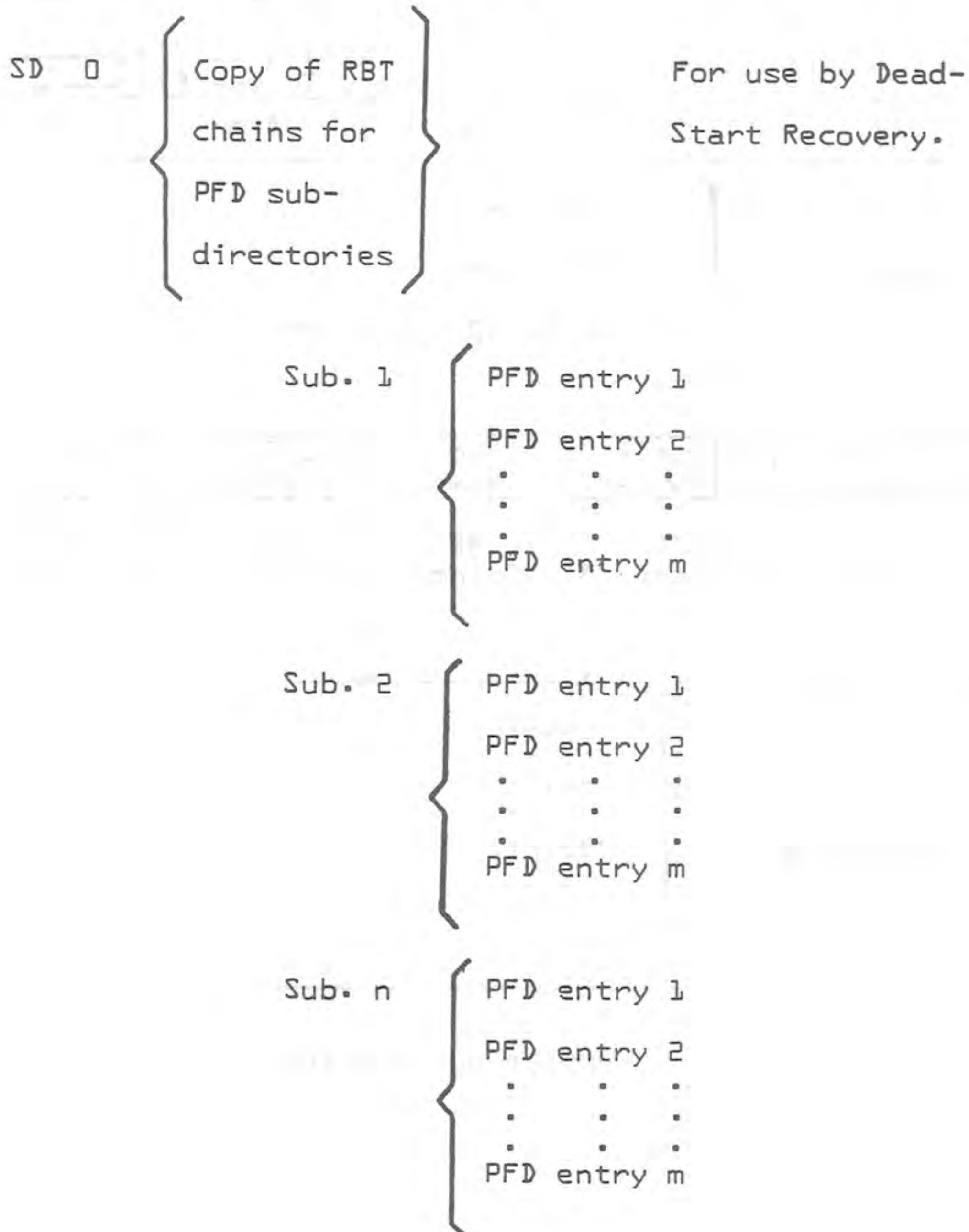


Figure (b)

PFD Entry

(normally 16 words)

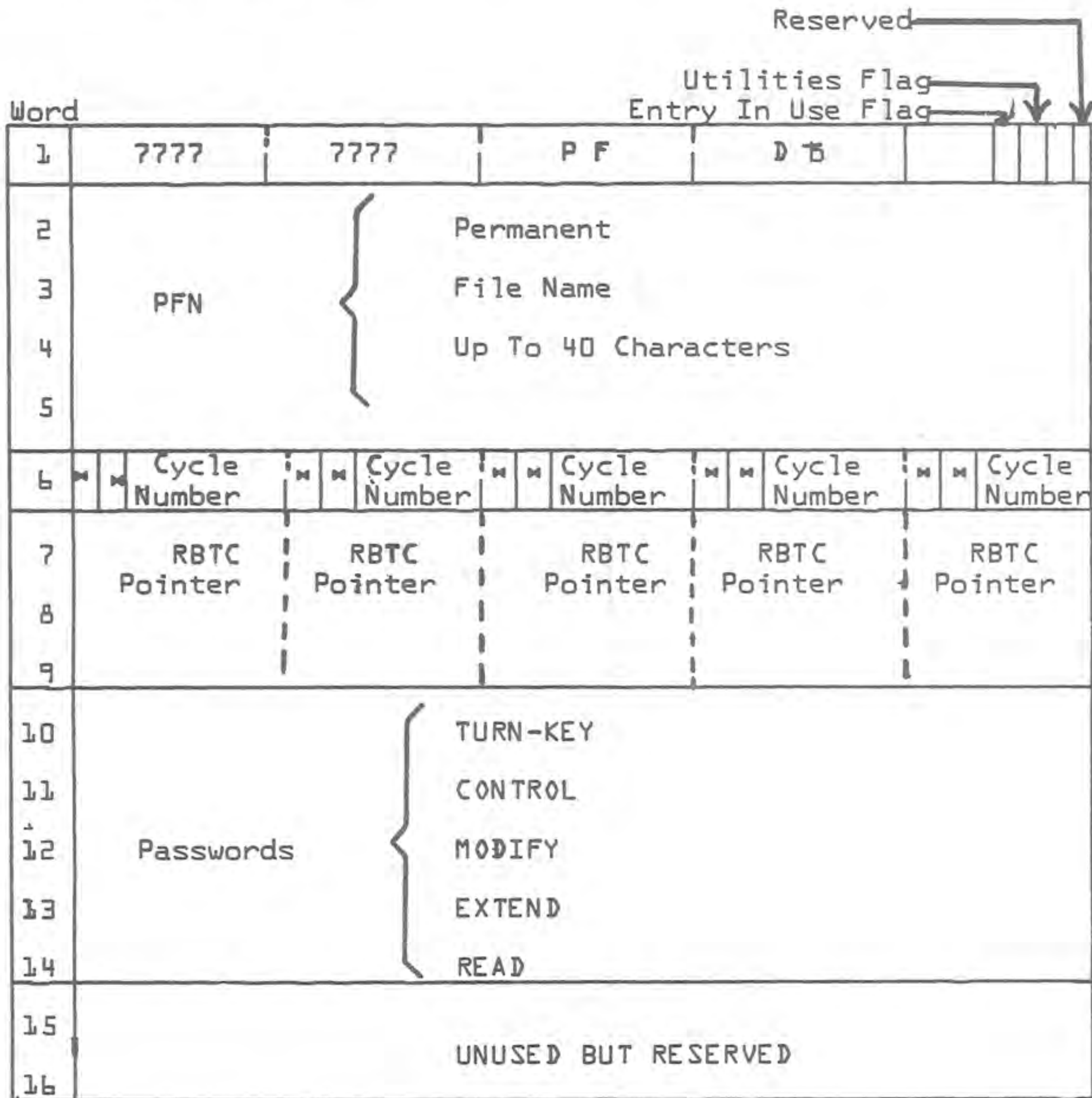


Figure {c}

THE RBT CATALOG

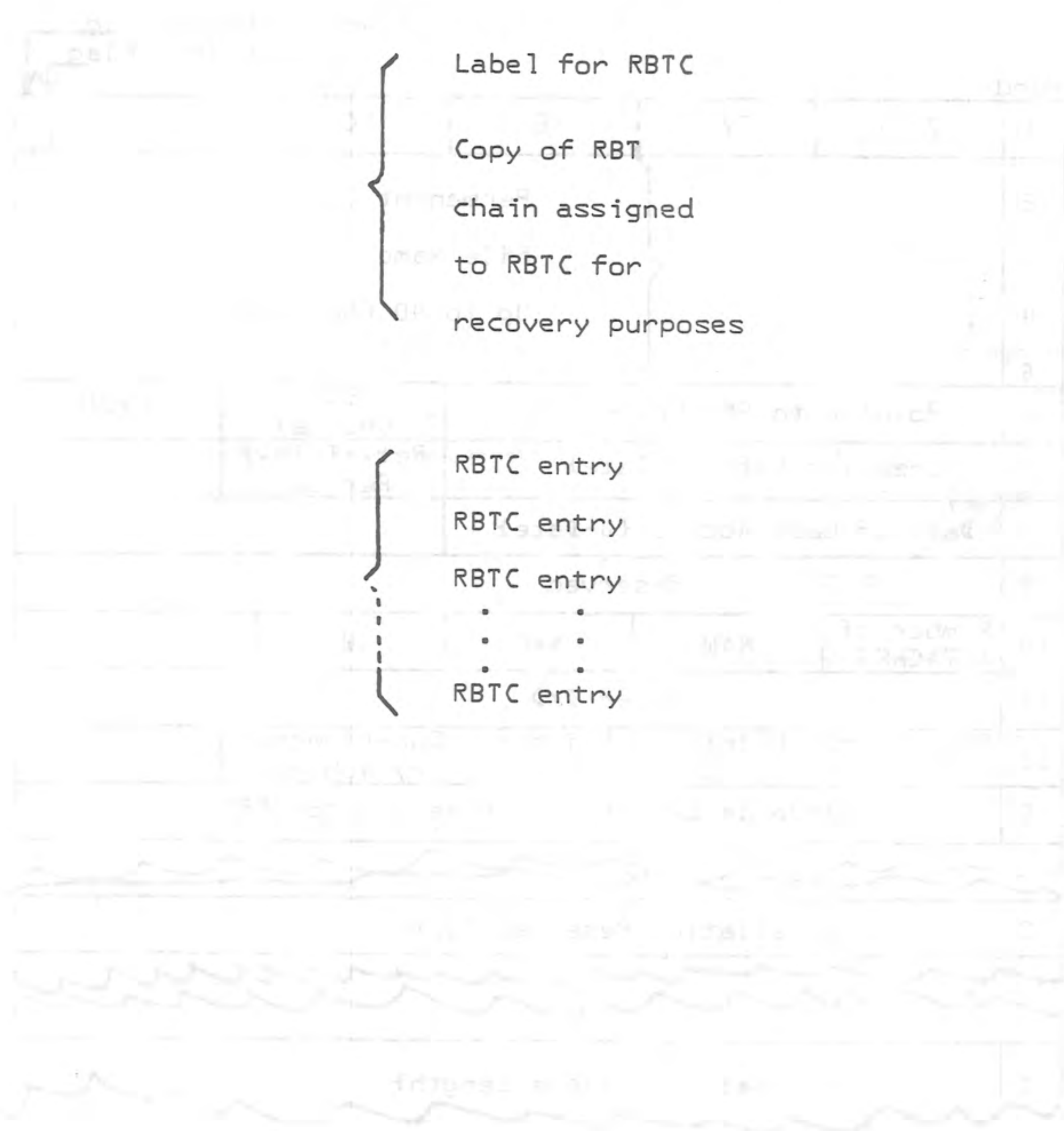


Figure {d}

RBTC ENTRY

{Variable Length}

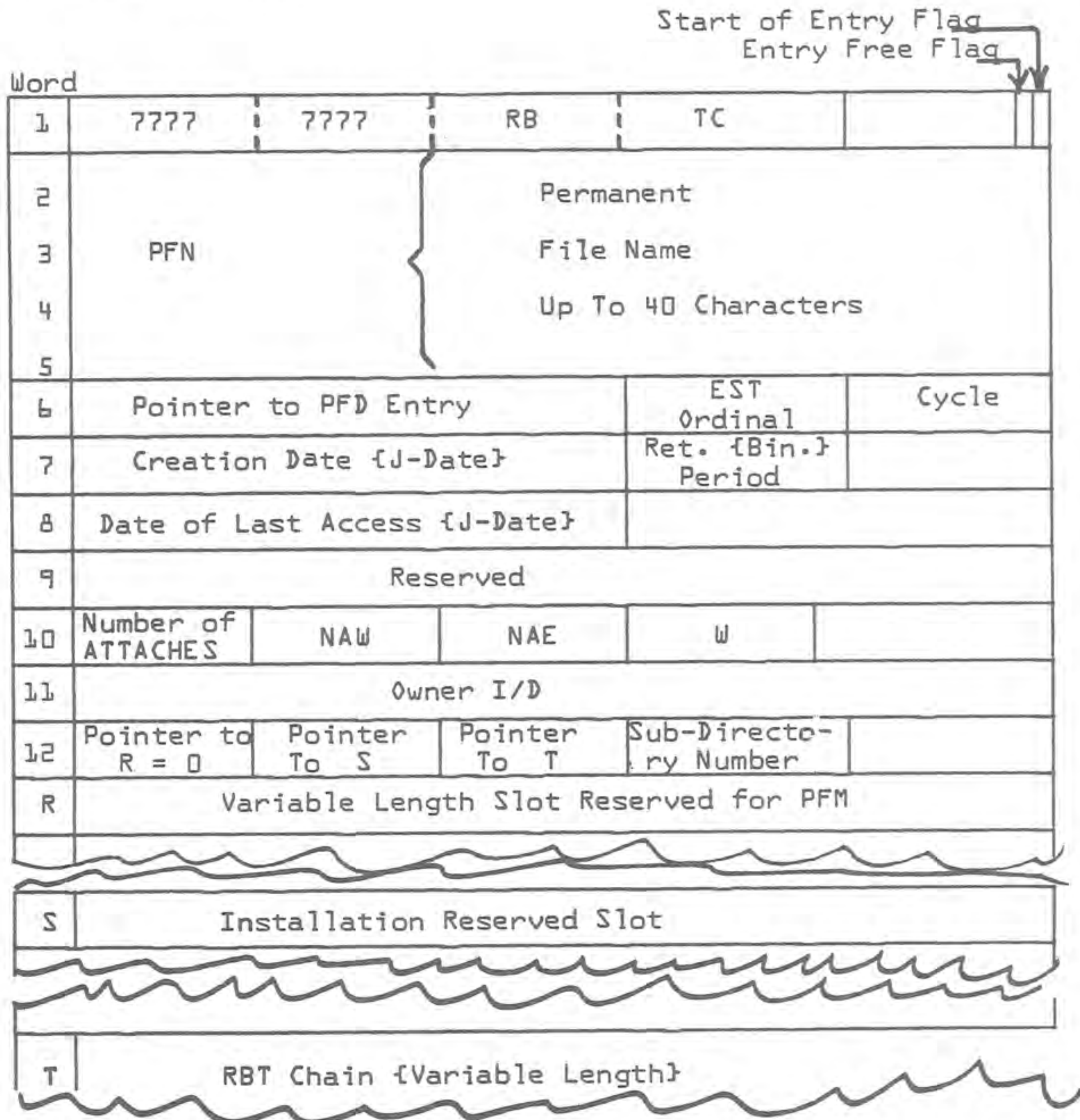


Figure {e}

APF TABLE ENTRY

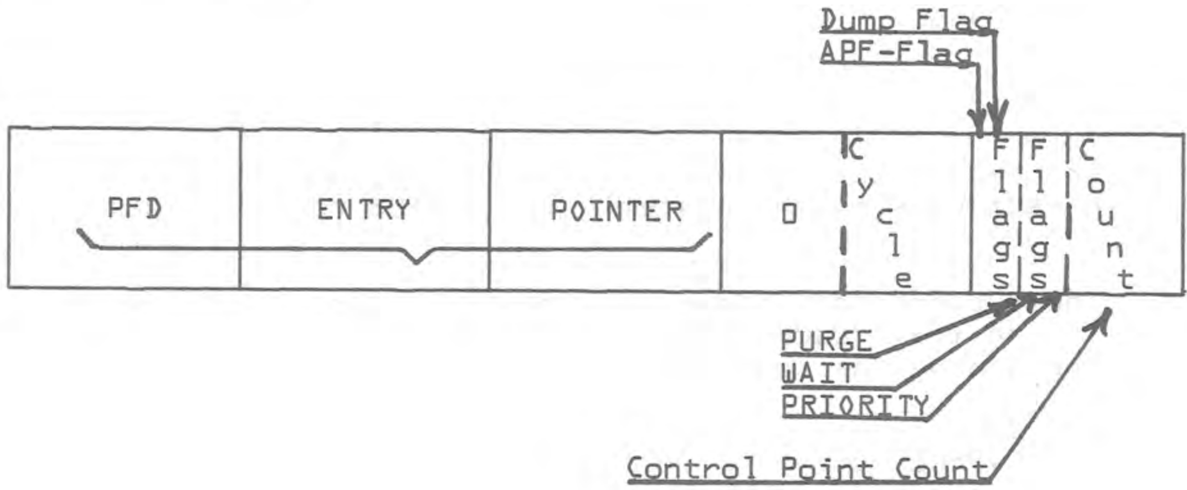
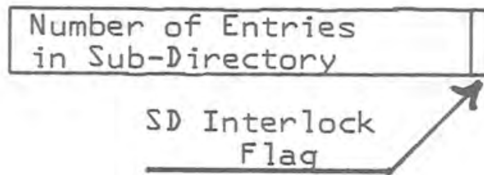


Figure {f}

SDT ENTRY

{1 CM Byte}



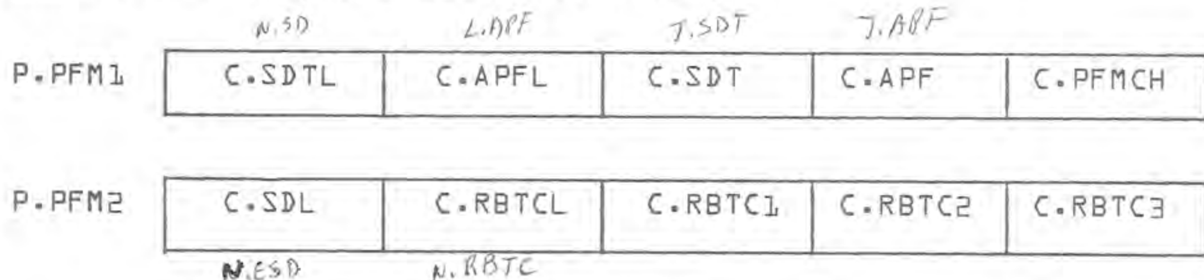
APF FLAG : Means when on, that PFM is in the process of changing information associated with this file and consequently the file is unavailable.

DUMP Flag : This is used by the dump utility and is put on when the entry refers to a dump program.

Entries in the PFD will be placed into appropriate subdirectories by the system. See SELSUB for the actual technique.

The subdirectory table (SDT) and its format will be shown in figure (f). Each entry in the SDT will consist of one byte containing the count and a one-bit interlock flag.

Two pointer words are used in low core. These are P.PFM1 and P.PFM2; they are illustrated below.



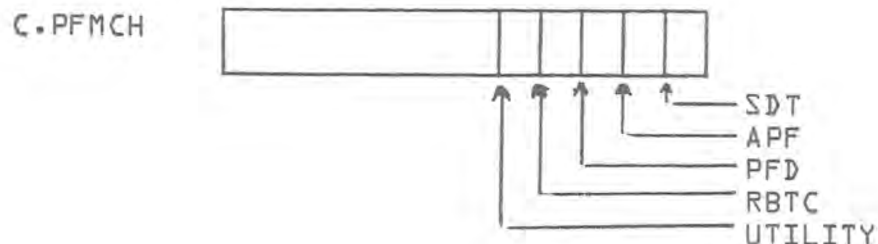
C.SDTL - This contains the value N.SD which is the number of subdirectories in the system. (Including dummy subdirectory zero).

C.APFL - This has the value of L.APF which is the length of the APF table in CM words.

C.SDT - This is the FWA of the subdirectory table.

C.APF - This is the FWA of the APF table.

C.PFMCH - This is the PF toggle byte used for subinterlocking disk and CM tables. It has the form -



SDT - interlock bit for SDT
APF - interlock bit for APF
PFD - interlock for new-name scanning
RBTC - interlock for RBTC
UTILITY- used when utility is running

C.SDL - This contains N.ESD which is the number of available slots in each subdirectory. (4 per PRU)

C.RBTCL - This contains the value N.RBTC which is the number of PRU's in the RBTC divided by 16.

C.RBTC1 - C.RBTC3 contain a pointer to the current end of information in the RBTC.

In addition the Permanent File Manager has reserved two words in each control point area. These are W.CPPF1 and W.CPPF2 and are described under the DELAY routine later in this chapter.

18.2 Functions

The initial complement of functions are four in number. These are CATALOG, ATTACH, PURGE, and EXTEND. In addition, an FDB MACRO is needed to set up the file data block.

FDB

The FDB is used to transmit information such as parameters and return codes, between the user and the PFM. It is generated automatically on a control card call but on a macro call the FDB must be generated with the system macro FDB, which has the following format:

```
fdbaddr  FDB  lfn, pfn, parameter list
```

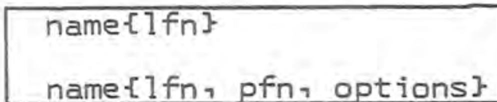
fdbaddr must be present in the location field. It is the location of word 5 of the FDB. The list must be preceded by a comma even if the PFN is not given and each parameter is separated by a comma. The entire list is terminated by a blank.

Parameters may include any of the following:

RP = retention period in days
PP = privacy procedure parameter
CY = cycle number
TK = turn-key password in a CATALOG function
PW = password list
CN = control password
MD = modify password
EX = extend password
RD = read password
SD = subdirectory
ID = user identification

Control Card Functions

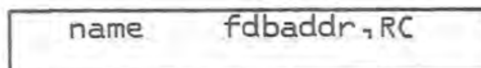
All control card requests share a common format:



where lfn is the first parameter identified by position. This name is associated with the fifth word of the FDB. The option list follows the parameter field. Options may be listed in any order, and they are separated by a blank or comma; the list is terminated with a right parenthesis. Each option is identified by the two character code listed in Section 2.2.

Macro Functions

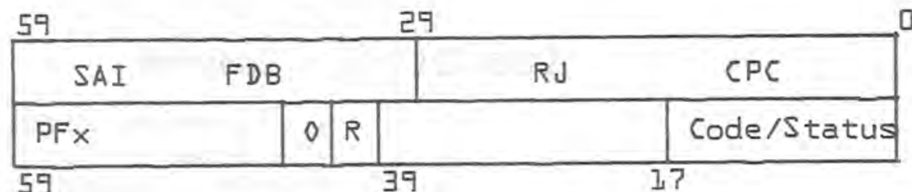
All macro requests share a common format:



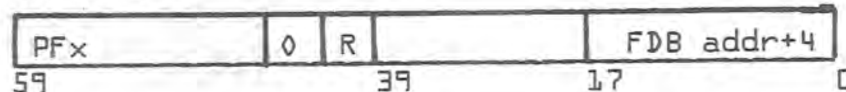
where fdbaddr is the location of the fifth word of a predefined FDB, and RC is an optional parameter that will return control to user on non-fatal error.

MACRO EXPANSION AND FDB FORMATS

Call to CPC



CPC Call to RA+1



FDB BLOCK

The FDB contains all parameters for PFM to execute the requested function. Its length will be determined by the number of parameters submitted. Minimum length is 6 words; thereafter, one word will be required for each parameter specified. The entire FDB must be terminated with a word containing zeros in bits 0→5.

Word	Entry	
1	PFN Up to	
2	40 Characters	
3		
4		17
5	LFN	Code/Status
6		
7	Parameters	
	Parameters	
		5
Last-word N	Blank	0

The 18-bit code/status field will be formatted as follows:

Bits 17 - 9: Error codes -- any (codes greater than 020_8 are fatal and abort the job). . .

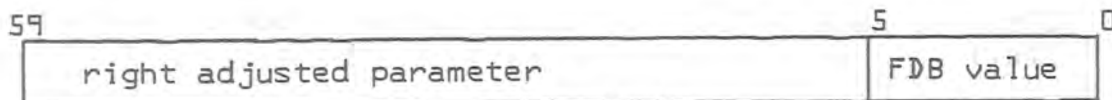
<u>User Return-Codes</u>

Code (Octal)	Meaning
0.	Function successful
1.	Format error
2.	LFN already assigned (ATTACH)
3.	LFN not found (CATALOG, EXTEND, PURGE)
4.	Blank PFN (CATALOG, ATTACH)
5.	Directory Full (CATALOG)
6.	Catalog Full (CATALOG, EXTEND)
7.	PF Device Unavailable
10.	Open Random File on a CATALOG, EXTEND
11.	Illegal Device for File Residence (CATALOG)
12.	ATTACH Request for Unknown File

Code (Octal)	Meaning
13	Cycle Reference does not exist (ATTACH, PURGE, EXTEND)
14	Invalid Cycle Number
15	Duplicate Name/Cycle or no slot on CATALOG
16	Attempt to Recatalog Existing Permanent File
17	Attempt to CATALOG Non-Local File
20	Function Attempted on Non-Permanent File
21	Function Attempted on Purged File
22	CATALOG Attempt, No Word Pairs
23	Cycle incomplete on an ATTACH
24	Duplicate ATTACH Request

Bits 8 - 0: PFM request code--an even number--stored by PFM when a request is issued. Completion of request is indicated by setting bit zero to one.

Parameter Words



As shown above parameters are stored, one per word in any order. FDB values are as below:

Octal Value	Meaning
00	end of FDB list
01	PP - privacy procedure parameter (display code)
02	RP - retention period in days (binary)
03	CY - cycle number (binary)
04	TK - turn-key password (display code)
05	CN - control password (display code)
06	MD - modify password (display code)
07	EX - extend password (display code)
10	RD - read password (display code)
11	unused
12	SD - sub-directory (Display Code)
13	null
14	ID - user identification
20-24	

All other values are reserved for future system use except 50_8 and above.

The PFM request codes are detailed below:

Octal Value	Function	
010	ATTACH	Return control to user on non-fatal error (if macro, call with ',RC')
020	CATALOG	
030	EXTEND	
040	PURGE	
110	ATTACH	Abort on any error (all other calls)
120	CATALOG	
130	EXTEND	
140	PURGE	

18.3 PFM Main Line

The PFM consists of one PP program for each function. These PP programs implement the functions requested by the CP job and transmitted via the FDB, as they are requested. They are PFC, PFA, PFP, PFE, corresponding to the functions CATALOG, ATTACH, PURGE, EXTEND respectively.

Only one copy of the PFM can be active per control point. In other words, PFM functions will be honored serially in requested order. To implement this on a macro call, CPC will turn off the CP when it receives a PF request and CPC will be recalled by a bit in the FDB when PFM has completed the function. CPC will then turn the CP back on and continue with the next function. (Compare same mode of operation of CIO.)

As noted in 3.5, it will be necessary to put the PFM into the delay stack as various hold-ups occur in satisfying function requests. The PFM will use two words in the control point area to store pointers while it is in the delay stack.

The entry routine to thePFM is shown elsewhere in the IMS. The crucial part is deciding, by examination of the Input Register, if this is a delay stack entry. If so, we have to restore the PFD and FDB information to the PP and return jump to progress with the request. On initial entry, the validity of the request is determined and control is then passed to the appropriate routines.

NOTES ON FUNCTIONAL IMPLEMENTATION AND CONTROL FLOW OF THE PFM

To implement multi-access for read and to save pending accesses, a system of waiting loops and queues is needed.

They are all coordinated by the APF table entry. The wait, priority and APF flags as well as the control point count all being involved.

Multi-read access is handled by "wait for access" routine. The detailed handling of these flags is illustrated elsewhere in the IMS. Note that as a prelude to the wait loops, a check is made on the APF flag. If the delay-loop is taken, it is necessary to ensure the entry is still there on exit. If it is gone, the calling function must re-search the APF table for the entry and re-issue the wait for access or take alternate action if the entry is not found in the table.

CATALOG (PFC)

The control flow is detailed elsewhere in the IMS. Reference is made to an Installation Parameter when the PFM is unable to make the correct directory entry for one of three reasons. These are: duplicate name, no spare cycle or no control permission when attempting to create a new cycle. In these cases, if the Parameter is zero, the user is aborted and if the Parameter is one, the system will attempt to honor the request by generating a new name for the file. Generation will be attempted by changing a character of the submitted PFN. The user must ensure a random file is closed before cataloguing. After an entry has been catalogued, it remains attached to that control point with all permissions in closed status.

An Installation Parameter defines the default retention period used, if one is not specified on a CATALOG. A period of 999 is taken as infinite.

Some general notes on file cycles follow: (See also Sect 18.1.7)
As a definition within this document, cycles of a file are taken as files sharing the same PFD entry and have the same names and passwords

Each cycle pointer in the PFD has its cycle number associated with it. As a cycle is purged, this number is made zero. On an ATTACH, the default cycle is the one with the largest cycle number, presumably the latest.

ATTACH (PFA)

The control flow for the ATTACH function is given elsewhere in the IMS.

EXTEND (PFE)

The control flow is detailed elsewhere in the IMS. This function, like PURGE, has only one parameter and that is LFN.

This implies that the file must be attached to the control point before these functions can be issued. The attach could have been achieved via the functions CATALOG or ATTACH (with, in this case, EXTEND permission).

The EXTEND function makes permanent, information added to the file after it has been attached, so that it will be recovered with the file in the future. The user must close a random file before issuing an EXTEND function on that file.

PURGE (PFP)

The control flow for this function is detailed elsewhere in the IMS.

As noted under EXTEND, this function requires only the LFN as a parameter. However, when the file was attached, it must have attached with control permission.

18.4 Routine Details

Section 18.5 - 18.9 contain the detailed information relating to the PFM routines. It is divided into five sections; the routines specific to each of the four PP programs and the routines that are common to several of them.

Index

Main line PFC 18.5.1
  Subroutines of PFC 18.5.2
Main line of PFA 18.6.1
  Subroutines of PFA 18.6.2
  Password Scheme 18.6.3
Main line of PFP 18.7.1
Main line of PFE 18.8.1
  Subroutines of PFE 18.8.2
Common routines 18.9.1

18.5 General

Name: PFC
Date: March 17, 1969
Version: 1

Purpose

PFC is that part of the Permanent File Manager that performs the CATALOG function. This results in the addition of a file to the Permanent File Directory and Catalog.

It has two modes of operation; an ▿initial▿ catalog and a ▿new cycle▿ catalog. The former occurs when the PFN is unknown to the system, and the latter occurs on the addition of a cycle to an already cataloged PFN.

Function

The main function performed by PFC are:

1. Common initialization
2. PFC initialization and parameter checking
3. Newcycle determination
4. PFD handling
5. RBTC handling
6. Termination

General Logic Flow

1. {PF} Common initialization
   PFC uses the common initialization routine contained in common deck INIT. This handles initial and delay stack entries to the function.

2. {CAT} PFC initialization
   {a} Check validity of submitted LFN.
   {b} Check it is not already permanent.
   {c} Check it of type local
   {d} Check it resides on valid mass storage
   {e} If file is active go to 2.{a}
   {f} Check at least one RBT word pair is assigned.
   {g} Ensure that if file is open, it is not a random file.

3. {CAT2} Newcycle determination
   {a} Extract and verify sub-directory if present in FDB.
   {b} If OK, search that sub-directory of file name
   {c} If not OK, or if not found, scan all sub-directories for file name.
   {d} If permanent file name not found, go to 4{a}
   {e} If in ▿newname▿ mode make up new name and go to 3{c}
   {f} Extract CY parameter from FDB, if present, and check its validity.
   {g} If not present and IP.RNF is not set, terminate. If IP.RNF is set, make up new name, set ▿newname▿ mode on and go to 3{c}.
   {h} Check PFD entry and ensure cycle number does not already exist.
   {i} Check that there is room for extra cycle. If none, go to 3{g}.
   {j} Ensure user has control permission.
   {k} Add new cycle number to PFD with incomplete flag on and write PFD entry out. Go to 5.

4. (CAT12) PFD handling
   (a) If "newname" mode write out the new name.
   (b) Select a sub-directory and reserve a slot in it.
   (c) Build the PFD entry from FDB data.
   (e) Set entry in use and incomplete flags then write PFD entry.

5. (CAT10) RBTC handling
   (a) Build an APF entry for the user's file. Set a pointer to the APF entry in the user's FNT entry. Set all permission bits on in the user's FST entry.
   (b) (CATA2) Select and read in the PRU from the Catalog (RBTC) to hold the new entry.
      (1) The FST entry for the Catalog is found and a dummy FST entry is built in the message buffer using the Catalog e.o.i. pointer from CMR (P.PFM2).
      (2) The e.o.i. PRU is read in. If the new entry fits wholly within the PRU, go to step 5.C. If not, the entry must start at the beginning of a new PRU (this tends to save read accesses by preventing small entries from spanning PRU's).
   (c) (CATA5) Build the new Catalog entry.
      (1) Build the header from internal (saved) items and from the FDB.
      (2) Append the RBT chain from CM and then add extra word pairs (to allow room for future file extensions).
   (d) (CATA20A) Update the e.o.i. pointer in CMR, then write out the new Catalog entry.

6. (CATA21) Termination

   (a) Reread PFD entry and insert RBTC entry pointer.
   (b) Clear incomplete flag and write entry back out.
   (c) If necessary, call installation privacy procedure.
   (d) Complete APF entry.
   (e) Terminate.

18.5.2 PFC Local Subroutines

<u>Name</u>

ALTER

<u>Function</u>

Shifts the permanent file name in item PFN right one character, end-off, and substitutes a random digit (0-7) as the left-most character.

<u>Entry Information</u>

PFN (20 bytes) contains the permanent file name.

Exit Information

PFN shifted and modified

Subroutines Called

None

Called By

PFC

Cells Changed

Direct (TEMP, D.T0 to D.T4)

Name

CHKPER

Function

Compares the passwords in the PFD entry against the password list in the FDB for Turnkey and Control passwords, i.e., Control permission (entered when user wishes to add a new cycle).

Entry Information

ENTCOUNT contains index to PFD entry in buffer (SECT1).

Exit Information

A = 0, passwords OK--user has control permission
A = minus, one or both passwords missing in
FDB--user does not have control permission

Subroutines Called

CHKPW

Called By

PFC

Cells Changed

None

Name

COPYPW

Function

Extracts a password from the FDB and stores it in the correct permission slot in the PFD entry (in SECT1). If the password is not in the FDB, a zero is stored in permission slot.

Entry Information

A-register contains the FDB octal code for the permission password sought.

Exit Information

None

Subroutines Called

EXFDB, MULT5, COPY

Called By

PFC

Cells Changed

Direct (TEMP1, TEMP2)
Other (BUF)

Name

CORBT

Function

Reads the RBT chain of a user's file and stores the number of CM words in the chain in COUNT.

Entry Information

D.FNT (5 bytes) contains the first FST word for the user's file

Exit Information

COUNT contains the number of CM words in the user's RBT chain.

Subroutines Called

None

Called By

PFC

Cells Changed

Direct (COUNT, D.T0 through D.T4)

Name

GETLFN

Function

Searches the FNT for a given local file name attached to user's control point. If name is not found, an error exit to the user occurs.

Entry Information

PFN+20 contains the seven-character local file name.
D.CPAD contains control point address.

Exit Information

A-register, D.T0 contains CM address of FNT entry.
D.FNT contains the FNT word for the found file.

Subroutines Called

SRCHCP, R.DFM, LFNDF, ERR

Called By

PFC

Cells Changed

Direct (D.T0 to D.T4, D.FNT to D.FNT+4)

Name

NEWAPF

Function

An open subroutine which finds an empty slot in the APF table, builds an APF entry and stores it in the slot. If no slot is available, a message is typed and after some delay (delay stack), the routine loops back and searches again for an empty slot.

Entry Information

POINT contains PFD pointer (3-bytes).
CYCLE contains PF cycle number.

Exit Information

APF contains index to new APF entry

Subroutines Called

RAPF, LFLAG, DAPF, BMES, DAPF, DELAY

Called By

PFC

Cells Changed

Direct {APF, D.T0 to D.T4}

Name

PUTCY

Function

Changes the Cycle parameter in the FDB to a null. If the cycle parameter is not found, an error exit to the user occurs.

Entry Information

None

Exit Information

None

Subroutines Called

EXFDB, FDBADR, ERR

Called By

PFC

Cells Changed

Direct {TEMP, D.T0 to D.T4}

Name

SELSUB

Function

An open subroutine which finds the least-full subdirectory by scanning the SDT. It then increments the SD entry count and tests it, putting out a warning message when it becomes 4/5 full or terminating the request if it is full {PFD full}. Finally SELSUB locates an empty entry slot in the subdirectory.

Entry Information

None

Exit Information

SUBD contains number of least-full subdirectory.
POINT contains a 4-byte pointer to an empty SD slot.
SECT1 contains the PRU with the empty SD slot.
ENTCOUNT contains the byte index of the empty SD slot.
SD interlock is set.

Subroutines Called

RSDT, LSDB, RSD, DELAY, LSDB, DIV5, RSDB, DSDT, PRNDF, R.DFM, ERR, SEARCH, BMES, DSD.

Called By

PFC

Cells Changed

Direct {SUBD, TEMP, TEMP1, TEMP2, D.T0 to D.T4, POINT to POINT+4, ENTCOUNT}
Other {NASLOT, SECT1}

Name

SRCHDR

Function

Examines the BUFFER to determine if the RBT header word is in the expected location. If it is not, an error exit to the user occurs

Entry Information

A = 0 look for an e.o.i. header.
A = 1 look for a data entry header.
RBTCIX contains the BUFFER index to the expected header position.

Exit Information

None

Subroutines Called

ERR

Called By

PFC

Cells Changed

Direct (D.T5, D.T6, D.T7)

18.6.1 General

Name: PFA
Date: 11 March 1969
Version: 1

Purpose

PFA is that part of the Permanent File Manager that performs the ATTACH function. This enables a user to reference a permanent file as though it were local to his control point.

Function

The main functions performed by PFA are

1. Common initialization.
2. PFA initialization and request validity checking.
3. Permission code generation.
4. Gain access to file and create APF entry.
5. Updates access information in RBTC.
6. Where needed reads RBT chains to high core and creates an FNT entry.

General Logic Flow

1. (PF) Common Initialization

   PFA uses the common initialization routine contained in common deck INIT. This handles initial and delay stack entries to the function.

2. (ATT) PFA Initialization

   (a) Check validity of submitted LFN.

(b) If valid sub-directory number was submitted, search that one for PFN, if not found and in other cases search all sub-directories.

(c) If not cound in any sub-directory, terminate with message.

(d) Extract and validate cycle parameter (if present) from FDB.

(e) If valid cycle parameter, scan PFD entry for its RBTC pointer. If found, go to 2(g); else terminate.

(f) If no cycle parameter, scan PFD entry for the RBTC pointer of largest cycle.

(g) Ensure selected cycle is complete and has not been dumped.

3. (ATT7) Permission code generation

(a) If IP.PP is set, extract PP parameter from FDB, verify validity and then call Installation Privacy Procedure. Go to 3(c).

(b) If IP.PP is not set, use subroutine CHPERM to generate permission.

(c) If no permission bits are set, abort.

4. (ATT82) Gain access to file

(a) Scan APF table for this cycle of the file. If not found, go to 4(c).

(b) Enter routine WFA (wait for access) described separately. Then go to 5(a) after setting "chain in use" mode.

(c) Create an APF entry for this cycle, and set off "chain in use" mode.

5. (ATT92) Update access information

(a) Read in first PRU of RBTC entry and verify it.

(b) Store current date as last access date and bump "attach" count.

(c) Backspace, then write RBTC entry back out.

6. Chain Handling

(a) If "Chain in core" mode, scan FNT for entry with

same APF pointer and save chain in core position. Go to 6(c)

(b) Copy RBT chain to high core requesting more word pairs via 1SX when needed and also reading subsequent PRU's of RBT entry when needed.

(c) Build FNT entry from available information. Save in it the APF pointer, the generated permission, the RBT chain position, and set security code to closed.

(d) Find FNT slot and copy entry to it.

(e) Terminate function

18.6.2 PFA Local Subroutines

Name

CHAPF

Function

Compares the contents of an APF entry contained in D.T0 to D.T4 against the parameter values in items POINT and CYCLE. If the parameters match the entry contents, the APF and PURGE flags are checked.

Entry Information

POINT contains a 4-byte PFD entry pointer.
CYCLE contains the cycle number of the user's file.
D.T0 to D.T4 contains an APF entry.

Exit Information

A = 1, entry OK.
A = 0, entry doesn't match parameters
A =-1, entry OK, except APF (entry interlock) flag is on.
A =-2, entry OK, except Purge flag is on.

Subroutines Called

None

Called By

PFA

Cells Changed

None

Name

CHPERM

Function

Compares the passwords in the PFD entry against those submitted by the user in the FDB. Permission is granted when passwords match or when no password is cataloged. If the user is granted no permissionsby this algotithm, a message is sent to the dayfile. (See password scheme in Section 18.6.3)

Entry Information

ENTCOUNT contains index to PFD entry in SECT1. BUF contains the 9-charactdr password name. SECT1 contains the user's PFD entry.

Exit Information

PERM contains the resultant permission codes, right-justified.

Subroutines Called

CHKPW, R.DFM

Called By

PFA

Cells Changed

Direct (D.Z3, D.Z4, PERM, TEMP)

Name

INAPF

Function

Examines the APF table to see if the entry for the user's file is there. On entry, a request is made to either examine a specific entry or the entire table. The entry is found but the file has been purged, an error exit to the user occurs.

Entry Information

APF = 0, scan entire table for users entry. APF interlock
is off.

A ≠ 0, examine entry indexed by value in item APF.
APF interlock is on.

Exit Information

A = 0, entry not found. APF interlock is off.
A =-1, entry found by APF (entry interlock) flag is on...
APF interlock is off.
A = greater than zero, entry found. APF interlock is on.

Subroutines Called

RAPF, LFLAG, DAPF, CHAPF, PFNDF, F.DFM, DSD, ERR

Called By

PFA

Cells Changed

Direct (APF, D.T0 to D.T4)

Name

WAIT FOR ACCESS

Function

This routine controls initial file accessing by forming queues; the APF entry flags are used for this prupose.

Any file is considered eligible for multi-read access when only read access has been granted. When successive eligible requests occur, multi-read access is initiated. (See Flow-Chart)

Entry Information

APF I/L must be on

Exit Information

Two EXITS:

1. Normal Exit is drop out of bottom of routine
2. Exception Exit occurs if file disappears from CM while queueing. This exits directly back to ATTACH main line.

Subroutines Called

BMES, LFLAG, DAPF, DELAY, RAPF, INAPF, ONWT

Called By

Entered serially by PFA main line

18.6.3 Password Scheme

5 Passwords altogether.
4 Associated with specific functions.
1 Is a turn-key to any function. {Optional}

Specific functions are:

CONTROL }
MODIFY }
EXTEND }
READ }

P/W's are specified at CATALOG time

The TURN-KEY	is	defined	as	TK=name
The CONTROL	is	defined	as	CN=name
The MODIFY	is	defined	as	MD=name
The EXTEND	is	defined	as	EX=name
The READ	is	defined	as	RD=name

All definitions are optional

Name consists of 1→9 characters {alphanumeric}

Order of definitions is not important.

User is given permission for any function for which no password was defined and also for any function for which he correctly submits the password.

EXAMPLE:

1. CATALOG e.g.
   TK=JIM,EX=JOHN,MD=MINE,RD=YES
   PW=JIM,JOHN {allows extend and control only}
   PW=JIM,MINE {allows modify, control}
   PW=JIM,YOUR,JOHN {allows extend and control}
   ↑invalid P/W ignored.
   PW=YES,JOHN {allows nothing}

2. CATALOG e.g.
   EX=JACK,CN=CONT
   PW=CONT {allows control, read, modify}
   PW=JACK {allows extend, read, modify}
   no p/w {allows read, modify}

3. CATALOG TK=MYFILE,EX=PASS
   e.g. PW=MYFILE {allows read, control, modify}
   PW=MYFILE,PASS {allows read, extend, modify, control}
   no p/w {allows nothing}

4. CATALOG MD=MINE
   e.g. PW=MINE {allows read, extend, modify, control}
   no p/w {allows read, extend, control}

18.7.1 General

Name: PFP
Date: 10 March 1969
Version: 1

Purpose

PFP is that part of the Permanent File Manager that performs the PURGE function. This result s in the deletion from the system of previously catalogued files.

Function

The main functions performed by PFP are:

1. Common initialization
2. PFP initialization and parameter checking
3. PFD handling
4. RBTC handling
5. Termination

General Logic Flow

1. {PF} Common Initialization
   PFP uses the common initialization routine contained in common deck INIT. This handles initial and delay stack entries to the function.

2. {PUR} PFP Initialization
   a. Check validity of submitted LFN
   b. Save APF pointer from FNT entry
   c. Check that file is permanent
   d. Check user has control permission.
   e. Ensure file has not already been purged

3. {PUR24} PFD handling
   a. Set purge bit in APF entry
   b. Pick up PFD pointer
   c. Determine sub-directory and get its interlock

d. Read PFD entry to PP.
e. Make sure cycle exists
f. Test that cycle is complete
g. Zero cycle number in PFD entry
h. Retrieve RBTC pointer
i. Save PF name
j. If last cycle, release PFD entry and decrement sub-directory count.
k. Write entry back out and drop sub-directory inter-lock.

4. {PUR50} RBTC entry handling
   a. If cycle was incomplete skip to {5}.
   b. Get RBTC inter-lock and read in first PRU of RBTC entry
   c. Check PFD pointer and cycle number.
   d. Set entry free flag
   e. Backspace and write entry back out
   f. Dump RBTC interlock.

5. Termination
   a. Load return code of zero
   b. Jump to exit routine

18.8.1 General

Name: PFE
Date: 28 February 1969
Version: 1

Purpose

PFE is a Permanent File program which implements a user request to append new information to an existing permanent file.

Function

The main functions performed by PFE are:

1. Validity checks the request
2. Validity checks the C.M. RBT chain of the user file against the cataloged {RBTC} chain.
3. Updates the Catalog entry to reflect the new information added to the user file.

General Logic Flow

1. {PF} Common Initialization

   PFE uses an initialization routine which is common to all P. F. programs and is part of the common deck INIT.

This routine is entered when the initial request is made and when a request is to continue after a visit to the delay stack.

2. (EXT) PFE Initialization
   a. Save the APF pointer from the user's FNT entry.
   b. Verify that the user's file is a permanent file.
   c. If the user's file is Random, verify that it has been closed.
   d. Save the P. F. permission codes from the user's FST entry.
   e. Verify that the user has extend permission for his file.
   f. Verify that the user's file has not been PURGED.
   g. Save the PFD pointer and the cycle number from the APF entry.

3. (SDSUBD) Access User File's PFD Entry
   a. Using the RBT address from the PFD pointer, search the FNT for the subdirectory containing the entry for the user's file.
   b. Save the subdirectory number.
   c. Read and validate the PFD entry.
   d. Save the RBTC pointer from the PFD entry.

4. (EXT12) Access the Catalog (RBTC) Entry
   a. Get the RBTC interlock.
   b. Save the RBT ordinal from the user's FNT entry.
   c. Save the e.o.f. pointer which is in the FNT entry for the RBTC.
   d. Build a dummy FST entry in the message buffer to be used in reading the Catalog.
   e. Read and validate the Catalog (RBTC) entry header.

5. (EXT18) Validate User's RBT Chain

   a. Compare the user's RBT chain in CM with the RBT chain in the Catalog entry.
   b. If all bytes in both chains are equal, go to step 7. (The last +1 PRU in byte C.RBTPRU may be different in two chains, but they are set equal prior to the compare.)
   c. If the chains are unequal, go to step 6. (The only legal difference is when the end of the cataloged chain is reached before the end of the CM chain.)

6. (EXT26) Transfer New RBT Bytes to Cataloged Chain

   When a Catalog entry is created, extra word pairs are included to allow for in-place expansion of the file. When the allotted space is exceeded as a result of EXTEND requests, a new entry must be created at the RBTC e.o.i. Exceptions occur when there is unused space following the entry or when the entry is already at the e.o.i.

a. Transfer the remainder of the CM RBT chain to the Catalog entry, a word-pair at a time.
b. If the space available for expanding the entry is exceeded before the CM chain has been transferred, a new Catalog entry must be created at the e.o.i. Go to step 8.
c. When the last CM word-pair is transferred, extra word-pairs are added to the end of the entry, if there is room.
d. If the entry being updated was at the e.o.i. and the e.o.i. word was overwritten, a new e.o.i. word is stored and the e.o.i. pointer in CMR (P.PFM2) is updated.

7. (EXT33) Complete the Entry Update, End PFE

a. If the Catalog entry spans PRUs, the current PRU is written and the PRU containing the entry header is reread.
b. Update the W-byte (entry length) and rewrite the header PRU.
c. Drop the RBTC interlock.
d. End of PFE, drop PP.

8. (EXT40) Initialize to Create a New Catalog (RBTC) Entry

To create a new Catalog entry at e.o.i., the old (source) entry, less the RBT chain, is copied to the Catalog e.o.i., then the RBT chain is appended from CM. Two buffers are required for the copy along with related RBTC pointers and PRU (buffer) indexes.

	Source Entry	New (e.o.i.) Entry
buffer	SECT1	SECT2
RBTC pointer	RBTC	EOIRBTC
PRU index	RBTCIX	ENTCOUNT

a. If the PRU containing the entry header is not currently in the buffer (entry spans PRU's), reposition and read it in. Initialize the PRU index (RBTCIX) and set the RBTC pointer (RBTC) for the next source PRU.
b. Set the "entry-free" flag in the header of the old entry.
c. Initialize the e.o.i. pointers (EOIRBTC, ENTCOUNT) and read the e.o.i. PRU.
d. If the new entry will not fit wholly in the remainder of the e.o.i. PRU, do one of two things: (1) If this is the last PRU of the Catalog, the EXTEND operation cannot be completed--drop the PP; (2) If this is not the last PRU, clear the e.o.i. word in

the Catalog and update the e.o.i. pointers so the new entry begins the next PRU.

9. (EXT45) Transfer Header and V-slots to New Entry

   a. If the source buffer is emptied before the transfer is complete, read in the next source PRU, and update the source pointers.
   b. If the e.o.i. buffer is filled before the transfer is complete, write the current PRU and update the e.o.i. pointers.
   c. If the Catalog is exceeded, terminate as in step 8(d)1.

10. (EXT50) Transfer the New RBT Chain from CM

   a. Move the user's RBT chain from CM to the e.o.i. buffer and add the extra word pairs to the end.
   b. If the e.o.i. buffer is filled before the transfer is complete, write the current PRU and update the e.o.i. pointers.
   c. If the Catalog is exceeded, terminate as in step 8(d)1.

11. (EXT54) Terminate the New Entry, End PFE

   a. Update the RBTC e.o.i. pointer in CMR (P.PFM2).
   b. Store the e.o.i. word in the buffer and write the new e.o.i. PRU.
   c. Drop the RBTC interlock.

12. (EXT57) Update the RBTC Pointer in the PFD Entry

   a. Get the Subdirectory interlock.
   b. Update the RBTC pointer for the correct cycle in the user's PFD entry.
   c. Drop the Subdirectory interlock.
   d. End of PFE; drop PP.

18.8.2 <u>PFE Local Subroutines</u>

<u>Name</u>

CKBUF

<u>Function</u>

Increments RBTCIX, the index for the SECT1 buffer, by 10 bytes (one RBT word pair). If the buffer is filled, it is written to the current position minus one PRU in the Catalog. Then the next PRU is read into the buffer.

Entry Information

None

Exit Information

None

Subroutines Called

CKBUFI, BACKSP, WRTPRU, READPRU

Cells Changed

None

Name

CKBUFI

Function

Increments RBTCIX ten bytes and tests for buffer-full. A system error results if the index value is odd.

Entry Information

None

Exit Information

A = 0, buffer is full.
A = minus, buffer not full.

Subroutines Called

ERRBTC

Cells Changed

Direct (RBTCIX)
Other (MULTIPRU)

Name

CKNXT

Function

Compares a word pair (from a Catalog entry) with a constant to see if it contains a Catalog header word.

Entry Information

TEMP2 contains the PP address of the word pair to be tested.

Exit Information

A = 0, header word found
A ≠ 0, No header found (or e.o.i. header found)
EOIHDR≠0, e.o.i. header found

Subroutines Called

MATCH

Cells Changed

Direct (TEMP1, TEMP2)
Other (EOIHDR)

Name

CLEARBT

Function

Clears the 10-byte buffer, RBTWRD, to zeros.

Entry Information

None

Exit Information

None

Subroutines Called

None

Cells Changed

D.Z1

Name

CLSECT2

Function

Clears the buffer, SECT2, to zeros.

Entry Information

None

Exit Information

None

Subroutines Called

None

Cells Changed

D.ZL

Name

ERRBTC

Function

Performs the normal error exit except that the RBTC interlock is dropped first.

Entry Information

None

Exit Information

None

Subroutines Called

DRBTC, ERR

Cells Changed

None

Name

ERRPFD

Function

Performs the normal error exit except that the subdirectory interlock is dropped first.

Entry Information

None

Exit Information

None

Subroutines Called

DSD, ERR

Cells Changed

None

Name

IXSET

Function

Multiplies value in fourth byte of pointer RBTC by five and stores result {byte index} in RBTCIX.

Entry Information

None

Exit Information

None

Subroutines Called

None

Cells Changed

RBTCIX

Name

MATCH

Function

Compares two PP fields, byte by byte, for equality.

Entry Information

TEMP1 contains address of first field.
TEMP2 contains address of second field
A-reg. contains number of bytes to compare.

Exit Information

A=0, fields are equal.
A≠0, fields are not equal.

Subroutines Called

None

Cells Changed

D.Z0, TEMP1, TEMP2

Name

SAVFST

Function

The dummy FST in the message buffer is accessed and the current Catalog position saved from it in the first three bytes of a four-byte pointer. The fourth pointer byte (PRU index) is set to zero.

Entry Information

A-REG contains address of pointer.

Exit Information

Pointer contains current Catalog position. The index byte is zero.

Subroutines Called

None

Cells Changed

Direct (D.Z1, D.FNT to D.FNT+4)

Name

SETFST

Function

Generates a dummy FST entry in the last two words of the message buffer. It uses a four-byte pointer to update an FST entry already formatted in D.FNT (10 bytes) and to set an RBTC entry index (RBTCIX or ENTCOUNT).

Entry Information

D.FNT (10 direct cells) contains the basic FST entry information.
A-register contains the address of a four-byte pointer to a Catalog entry.

Exit Information

Message Buffer +4 contains the dummy FST. RBTCIX contains the index to the entry within the Catalog PRU.

Subroutines Called

None

Cells Changed

Direct (D.Z0, D.Z1, D.T2 to D.T4, RBTCIX or ENTCOUNT)
Other (REQTQB+1)

Name

SFSUBD

Function

An open subroutine which searches the FNT for an entry with a given RBT address value in C.FFRBA.

Entry Information

POINT (a direct cell) contains the RBT address to be searched for.

Exit Information

SUBD (a direct cell) contains the subdirectory number.
D.FNT (5 direct cells) contains first FST word for subdirectory.
D.T0 contains the FST address for subdirectory.
A=0, FNT entry found.
A≠0, FNT entry not found.

Subroutines Called

SDSRC

Cells Changed

Direct (SUBD, COUTN, D.T0, D.FNT to D.FNT+4)

Name

UPDEOI

Function

Updates the e.o.i. pointer in CMR (P.PFM2) for the Catalog

Entry Information

EOIRBTC contains the current RBT address, ordinal and PRU number for the Catalog. RBTCIX contains the word index within the PRU.

Exit Information

None

Subroutines Called

DIV5

Cells Changed

D.T0 to D.T4

18.9.1 Common Routines

Name

BACKSP

Function

Backspace one PRU

Entry Information

FST address in REQTAB+1
Flags = 0400 (FST Present)
JTEMP3 = First RBT ordinal

Exit Information

FST Information points to beginning of PRU, these pointers are in the address given in the entry information.

Subroutines Called

PFDIO, R.EQEQS

Called By

PFC, PFA, PFP, PFE

Cells Changed

STACKRE1, STACKRE2, D.T0 to D.T7

Name

BIND

Function

Convert a binary number to three display code, decimal characters.

Entry Information

Binary number in SUBD.

Exit Information

Characters in 18 bits of ACCUM.

Subroutines Called

DIV5

Called By

PFNDF, ERR

Variabled Used

TEMP1, TEMP2, REMAIN, SCRATCH, SUBD

Cells Changed

TEMP1, TEMP2, REMAIN, SCRATCH

Name

BMES

Function

Write message to B-display

Entry Information

ADDR. of message in ACCUM.

Exit Information

None

Subroutines Called

None

Called By

PFC, PFA

Cells Changed

D.T0 to D.T4

Name

CHKCY

Function

To check occurence of cycle numbers in PFD entry

Entry Information

ACC = cycle number
PFD entry in SECT1

Exit Information

ACC -VE	DUPLICATE
ACC=0	No more slots
ACC +VE	OK. (=slot position 1-5)

Subroutines Called

None

Called By

PFC, PFA, PFE

Cells Changed

TEMP, SCRATCH

Name

CHKPW

Function

Routine checks if named P/W is in FDB

Entry Information

ACC had P/W position within PFD entry

Exit Information

ACC = 0 If OK
ACC =-1 If not OK

Subroutines Called

COMPARE, EXFDB

Called By

CHKPER, CHPERM

Cells Changed

TEMP1, REMAIN

Name

COMPARE

Function

Compares two nine-character fields

Entry Information

TEMP1 has address of first field
BUF is second field

Exit Information

ACC = 0 Same
ACC =-1 Differ

Subroutines Called

None

Called By

CHKPW

Cells Changed

TEMP2

Name

COPY

Function

PP storage move

Entry Information

ACC = No. of Bytes
TEMP1 = Source Address
TEMP2 = Target Address

Exit Information

None

Subroutines Called

None

Called By

PFC, PFA, PFP, PFE

Cells Changed

TEMP, TEMP1, TEMP2

Common Routine

Name

CYCLIC

Function

Scans all of PFD for a PFN. If found, it will print its location to the dayfile except when there is only one subdirectory or when in "NEWNAME" mode.

Entry Information

On CATALOG PFD I/L should be on
On ATTACH PFD I/L is not used

Exit Information

If found, drops out bottom of code with
SD I/L ON.
If not found, branches to 'CY3' with
SD I/L OFF.

Subroutines Called

RSD, DELAY, SEARCH, BIND, PFNDF, R.DFM., RSDT, DSDT

Called By

PFC, PFA

Variables Used

SUBD, NASLOT, NWNAME, CYCLE

Name

DAPF

Function

DROP APF I/L

Entry Information

None

Exit Information

None

Subroutines Called

None

Called By

INAPF

Variables Used

None

Cells Changed

D.T0 through D.T4

Name

DBIN

Function

Convert three display coded digits to binary

Entry Information

Digits in BUFFER+3 and upper bits of BUFFER+4

Exit Information

Number in ACCUM

Subroutines Called

MULT5

Called By

MAIN LINE

Variables Used

TEMP

Calls Changed

TEMP

Name

DELAY

Function

Stores essential information from PP memory to C.PT area necessary for normal continuation after coming out of delay stack. Should only be called from Main Line, (unless TIME is less than 10).

Entry Information

Normal Entry (DELAY): Delay time in ▿TIME▿ entry by RJM.

Other Entry (DELAYP): Delay time in ▿TIME▿ entry by LJM. Information in C.PT. area is not updated.

Exit Information

If ▿TIME▿ is greater than LIMIT program is put into delay

stack; else delay is made locally in both cases control is eventually passed to CALLING routine at next instruction.

Subroutines Called

R.PAUSE, ERR, R.MTR

Called By

PFC, PFA, PFP, PFE

Variables Used

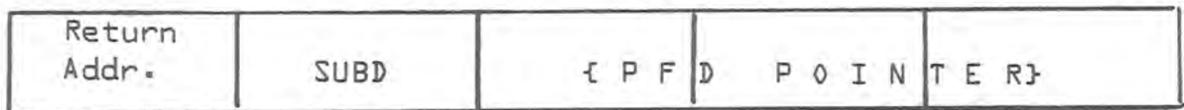
TIME, SUBD, POINT, CYCLE PERM, APF, NWNAME, RBTC

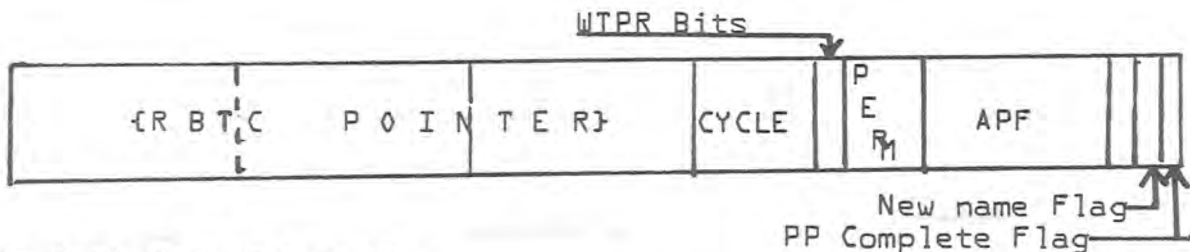
Cells Changed

D.T0 through D.T4

PFM use of C.PT area consists of two words. (Written by DELAY, read back by INIT.)

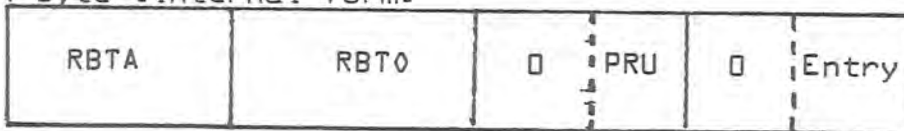
W.CPPF1





PFM Disk Ponter Format
3 Byte (external form)

4 Byte (internal form)



Name

DIV5

Function

Divide number in ACCUM by five

Entry Information

Number in ACCUM

Exit Information

Answer in ▿SCRATCH▿
Remainder in ▿REMAIN▿

Subroutines Called

None

Called By

LSDB, RSDB, BIND, PFC, PFE

Variables Used

SCRATCH, REMAIN

Cells Changed

SCRATCH, REMAIN

Name

DPFD

Function

Drop PFD I/L

Entry Information

None

Exit Information

None

Subroutines Called

None

<u>Called By</u>

CATALOG

<u>Variables Used</u>

None

<u>Cells Changed</u>

D.T0 to D.T4

<u>Name</u>

DRBTC

<u>Function</u>

Drop RBTC I/L

<u>Entry Information</u>

None

<u>Exit Information</u>

None

<u>Subroutines Called</u>

None

<u>Called By</u>

CATALOG

<u>Variables Used</u>

None

<u>Cells Changed</u>

D.T0 to D.T4

<u>Name</u>

DSD

<u>Function</u>

Drop SD I/L

Entry Information

Sub-directory in SUBD

Exit Information

None

Subroutines Called

LSDB, RSDB

Variables Used

None

Cells Changed

None

Name

DSDT

Function

Drop SDT I/L

Entry Information

None

Exit Information

None

Subroutines Called

None

Variables Used

None

Cells Changed

D.T0 to D.T4

Name

EORBTC

Function

Checks the saved RBTC EOF pointers of RBTC file against the current position shown in dummy FST in last two words of PP message buffer.

Entry Information

ENDRBTC contains RBTC EOF pointer and dummy FST for RBTC must be in message buffer.

Exit Information

ACCUM = 0, EOF reached, last PRU has been written

Subroutines Called

None

Called By

PFC, PFE

Variables Used

D.T0 to D.T4, D.Z1

Name

ERR

Function

Error Processor. Error codes noted in listing.

Entry Information

ACC+VE: System error message written with code in ACC. to D/F. Then aborted.

ACC-VE: User error, message should already have been given so just abort, if -20_8. Else return error code.

Exit Information

If ACC = 0: Error Flag on

Subroutines Called

BIND

Called By

PFC, PFA, PFP, PFE

Variables Used

SUBD

Cells Changed

SUBD

Name

EXFDB

Function

Extract parameter of given FDB - value from FDB.

Entry Information

Value in ACCUM

Exit Information

ACC = -1 If not found
ACC else is position found and parameter is in BUFFER -
BUFFER+4

Subroutines Called

FDBADR

Called By

PUTCY, PFC, PFA

Variables Used

SCRATCH, TEMP, BUFFER

Cells Changed

SCRATCH, TEMP

Name

FDBADR

Function

Calculates address of FDB+4 and tests its validity.

Entry Information

None

Exit Information

Normal exit - 18 bit address in ACCUM
Error exit - system code 1 if address of range

Subroutines Called

R.TFL, ERR

Called By

PFC, PFA, PFP, PFE

Variables Used

FDB

Name

FINDRBR

Function

Locate RBR Table

Entry Information

D.T1 contains RBR ordinal
D.Z+D.Zt contain P.RBR/P.RBT

Exit Information

ACCUM = RBR header word address

Subroutines Called

None

Called By

PFDIO, PFC

Variables Used

D.T1, D.Z1+C.RBRAD, D.Z2+C.RBRAD

Cells Changed

D.T7

Name

GETRBT

Function

Read in the next RBT owrd pair from CM into RBTWRD (10 bytes).

Entry Information

The first byte of RBTWRD must be set prior to the first call to GETRBT

Exit Information

None

Subroutines Called

None

Called By

PFC, PFE

Variables Used

PRBT

Name

INIT

Function

Initialize PF routine for either initial entry or Delay Stack Return

Entry Information

Input register contains FDB address. If bit 41 is set, implies delay entry.

Exit Information

On initial entry, none if OK, else aborts on delay entry, restores critical pointers and jumps to continue processing

Subroutines Called

R.PAUSE

Called By

PFC, PFA, PFE, PFP

Name

IOPFD

Function

Uses when reading or writing a PRU to or from a subdirectory.

Entry Information

Order in accumulator to read or write

Exit Information

None

Subroutines Called

PFDIO

Called By

READPFD, WRTPFD

Name

JPNCK

Function

Check file name in PFN with file name in SECT1

Entry Information

PFN, SECT1, A-reg = Entry Count

Exit Information

A ≠ 0 No comparison
A = 0

Subroutines Called

None

Called By

SRPRU

Variables Used

SECT1, PFN

Cells Changed

D.T0, D.T1

Labels JK20

Name

LFLAG

Function

Load flag byte of current APF entry

Entry Information

APF entry pointer in APF

Exit Information

Byte in ACCUM
Address in SCRATCH for writing back out.

Called By

INAPF, PFC, PFA, PFP, PFE

Variables Used

APF, SCRATCH

Cells Changed

SCRATCH, D.T0 to D.T4

Name

LFNDF

Function

Writes LFN to job dayfile

Entry Information

FDB header in PFN (5 words)

Exit Information

None

Subroutines Called

R.DFM

Called By

PFC, PFA, PFP, PFE

Name

LSDB

Function

Load SDT byte of specified sub-directory

Entry Information

Sub-directory in SUBD

Exit Information

Byte in ACCUM.

Subroutines Called

DIV5

Called By

RSD, DSD

Variables Used

SUBD, SCRATCH, REMAIN

Cells Changed

D.T0 to D.T4

Name

MULT5

Function

Multiply by five

Entry Information

Number in ACCUM

Exit Information

Result in ACCUM

Subroutines Called

None

Called By

COPYPN, PFC, PFA, PFP, PFE

Variables Used

REMAIN

Cells Changed

REMAIN

Common Routine

Name

NEWAPF

Function

Creates an APF entry

Entry Information

POINT, CYCLE

Exit Information

APF has entry POSITION

Subroutines Called

RAPF, LFLAG, DAPF, R.DFM, DELAY, BMES

Called By

PFC, PFA

Name

PFDIO

Function

Set up stack request to read or write one PRU

Entry Information

ORDER = Order code
FLAGS = FNT, FET Flag
PSEUBF = Address of Buffer

Exit Information

STACKRE1, STACKRE2 set up

Subroutines Called

FIND RBR

Called By

WRIPFD, SEARCH, WRTRBT

Variables Used

ORDER, FLAGS, PSEUBF, P.RBT, D.FNT+C.FFRBA

Cells Changed

STACKRE1, STACKRE2, TEMP, D.Z1 to D.Z5, D.T0 to D.T4

Labels

None

Name

PFNDF/PFNDFZ

Function

Writes current cycle and PFN to Dayfile. If cycle is zero, prints a double asterisk instead.

Entry Information

PFN in PFN
CYCLE in CYCLE

Entry1: PFNDF--MESSAGE TO JOB AND SYSTEM DAYFILES
Entry2: PFNDFZ--MESSAGE TO JOB DAYFILE ONLY

Exit Information

None

Subroutines Called

BIND, R.DFM

Called By

PFC, PFA, PFP, PFE

Variables Used

SUBD, TEMP1

Cells Changed

TEMP1

Name

RAPF

Function

Request APF table I/L

Entry Information

None

Exit Information

None

Subroutines Called

RQMSK, DELAY

Called By

INAPF

Variables Used

SCRATCH

Cells Changed

SCRATCH

Name

READCAT

Function

Used to read one PRU of RBTC to PP buffer

Entry Information

None

Exit Information

PRU in SECT1

Subroutines Called

SRCHFNT, ERR, READPRU

Called By

PFA, PFP

Name

READPFD

Function

Reads PRU from a sub-directory to SECT1

Entry Information

Same as WRTPFD

Exit Information

PRU in SECT1, sub-directory interlock off

Subroutines Called

IOPFD, R.READP, MULT5

Called By

PFC, PFP, PFE

Name

READPRU

Function

Reads a PRU

Entry Information

REQTAB {three words for request stack.
JTEMP3 must have first RBT word pair ordinal
FLAGS must contain fourth byte for request, word two
PSEUBF must have address of first PP word.

Exit Information

PRU in BUFFER

Subroutines Called

PFDIO, R.READP

Called By

PFC, PFA, PFP, PFE

Variables Used

ORDER, REQTAB, STACKREL

Name

REND

Function

Write toggle byte back out

Entry Information

Toggle byte in ACCUM.
Pseudo channel I/L on.

Exit Information

I/L now off

Subroutines Called

R.DCH

Called By

RAPF, RSDT

Variables Used

None

Cells Changed

D.T0 to D.T4

Name

REQMSK

Function

Used in requesting of sub-interlocks by all PFM routines.

Entry Information

Mask in accumulator

Exit Information

ACCUM zero if OK
ACCUM-VE if not obtained

Subroutines Called

RS, REND, R.DCH

Called By

PFC, PFA, PFP, PFE

Name

RPFD

Function

Request PFD I/L (to coordinate duplicate name scanning)

Exit Information

ACC = 0 If OK
ACC =-1 If unsuccessful

Subroutine Called

REQMSK

Called By

PFC

Variables Used

SCRATCH

Cells Changed

SCRATCH

Name

RBTC

Function

Request RBTC interlock

Entry Information

ACC = 0 if OK
ACC =-1 if unsuccessful

Subroutines Called

REQMSK

Called By

PFC, PFA, PFP, PFE

Name

RS

Function

Read in toggle byte

Entry Information

None

Exit Information

Toggle byte in ACCUM and SCRATCH pseudo channel. I/L is on.

Subroutines Called

R.RCH

Called By

RAPF, RSDT

Variables Used

SCRATCH

Cells Changed

SCRATCH, D.T0 to D.T4

Name

RSD

Function

Request I/L on a sub-directory

Entry Information

Sub-directory in SUBD

Exit Information

ACC = 0 If OK.
ACC =-1 If unsuccessful

Subroutines Called

RSDT, LSDB, RSDB, DSDT

Variables Used

SCRATCH

Cells Changed

SCRATCH

Name

RSDB

Function

Restore sub-directory byte

<u>Entry Information</u>

Byte in ACCUM. Sub-directory in SUBD.

<u>Exit Information</u>

None

<u>Subroutines Called</u>

DIV5

<u>Called By</u>

RSD, DSD

<u>Variables Used</u>

TEMP, SUBD, SCRATCH, REMAIN, TEMP1

<u>Cells Changed</u>

TEMP, SCRATCH, REMAIN, TEMP1, D.T0 to D.T4

<u>Name</u>

RSDT

<u>Function</u>

Request SDT I/L

<u>Entry Information</u>

None

<u>Exit Information</u>

None

<u>Subroutines Called</u>

REQMSK, DELAY

<u>Variables Used</u>

SCRATCH

<u>Cells Changed</u>

SCRATCH

Name

SDSRC

Function

Search FNT for subdirectory entry

Entry Information

Subdirectory number in SUBD

Exit Information

A ≠ 0 Not found
A = 0 D.FNT - D.FNT+4 Contain FST word 1
       D.RA = FST address
       D.TO = FNT address

Subroutines Called

SRCHFNT, BIND

Called By

SEARCH, IOPFD

Variables Used

JSDT, +1, +2, SUBD, D.TO

Cells Changed

D.FNT-D.FNT+4, D.FA

Name

SEARCH

Function

Search a given subdirectory for given file name or an empty slot

Entry Information

NASLOT = 0, Look for SLOT
       = 1, Look for NAME

Exit Information

POINT+3 - Negative, Not found

If found
PFD pointer in POINT,+1,+2,+3
{POINT+3 is no. of CM of DISP within SECT1}
Entry count in ENTCOUNT

Subroutines Called

SDSRC, PFDIO, SRPRU, RSD, DSD

Called By

SELSUB, CYCLIC, PFC, PFA

Variables Used

JSDT, P.FA, SECT1, NASLOT, NSD

Cells Changed

COUNT, TCOUNT, REQTAB, FLAGS, ORDER, PSEUBF, D.T0 to D.T4, POINT, +1, +2

Labels

SEA05, SEA20, SEA27, SEA30, SEA35

Name

SRCHCP

Function

Search FNT for LFN at this control point.

Entry Information

A-register contains addressof LFN in PP

Exit Information

A = 0 is a match
A ≠ 0 no match
D.T0 is addr. of FNT entry

Subroutines Called

SRCHFNT

Called By

PFC, PFA, PFP, PFE

Name

SRCHFNT

Function

Search FNT for a given file name

Entry Information

A-register contains address of name to match

Exit Information

A = 0 Match
A ≠ 0 No match
D.T0 is address of FNT entry

Called By

SDSRC, PFC, PFA, PFE

Variables Used

P.FNT, LE.FNT

Cells Changed

D.FNT to D.FNT+4, D.Z1, D.Z2, D.T0 to D.T4

Labels

SERCHLOP, GETNEXT

Name

SRPRU

Function

Search a PRU for a name or a slot

Entry Information

A = 0 Search for slot
A ≠ 0 Entry count in ENTCOUNT, POINT+3
Entry free flag set to 1

Exit Information

None

Subroutines Called

JPNCK

Called By

SEARCH

Variables Used

SECT1

Cells Changed

D.T2, ENTCOUNT, POINT+3

Labels

SRP1, SRP6, SRP7, SRP8

Name

UPDWC

Function

Update word count. Used by PPRES during non-stop disk reading.

Entry Information

None

Exit Information

None

Subroutines Called

None

Called By

PPRES (PFC, PFA)

Name

WRTPRU

Function

Writes a PRU to the file described as in READPRU

Entry Information

See READPRU

Exit Information

See READPRU

Subroutines Called

PFDIO, R.WRITEP

Called By

PFC, PFA, PFP, PFE

Name

WRTPFD

Function

Write one PRU of a subdirectory

Entry Information

RBTA in POINT
RBTØ in POINT+1
PRU in POINT+2

Exit Information

One PRU is written

Subroutines Called

PFDIO, RSD, DSD, R.WRITEP, SDSRC

Called By

PFC, PFP, PFE

Variables Used

SECT1, STACKREL, 2, POINT

Cells Changed

ORDER, REQTAB, +1, +2, FLAGS, PSEOBF

18.10.0 System Interface

Mods to:

- Stack Processor
- REQ
- 4ES
- 1AJ
- CTS
- CIO
- OPE
- CLO
- 1EJ
- DSD

New progs:

- PFCCP
- ACE
- 6PC

Non-stop Read (Stack Processor, PP resident)

RSM 1606

This function is not to be confused with READNS. It is a special communication between the Stack Processor and the permanent file manager. It provides an estimated 30 to 1 increase in the rate at which we can sequentially scan the P. F. Directory.

Request for a Specific Mass Storage Device (REQ, 4ES)

RSM 1766

REQ
This routine has been modified to accept and pass on to 4ES (in FST byte C.FLRBEB), a request for a specific mass storage device, by EST ordinal, instead of just a device type. This change was requested so the Load utility program could restore permanent files to the devices from which they were dumped whenever possible. (Particularly important when restoring a device dump).

4ES
This routine interprets the FST information and, if an EST ordinal is found (fourth byte), attempts to assign the file to that device. If it cannot, the assignment is made by device type only.

P. F. Control Cards (1AJ, PFCCP, ACE)

RSMs 1764, 1767, 1769

Call to PFCCP (1AJ)
On detecting a request for one of four Permanent File func-

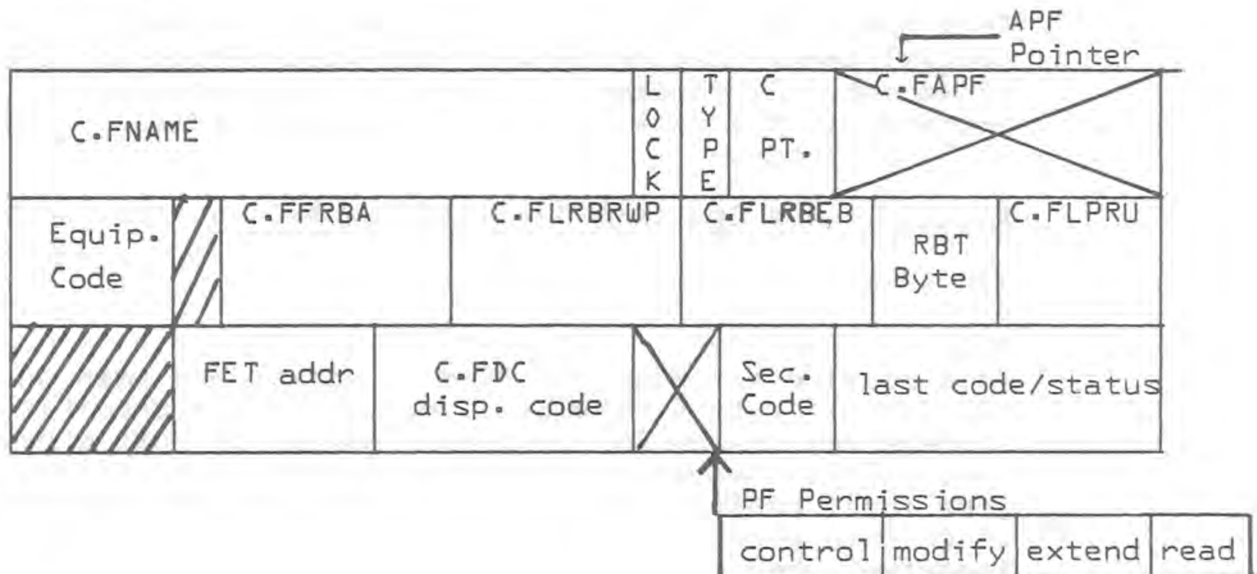
tions, 1AJ calls PFCCP to the control point and transfers control. 1AJ will not pass these cards (containing passwords) to the dayfile.

PFCCP
All permanent file control cards are processed by this program. The control card image is in RA+70B. The program processes the control card, builds an FDB and calls the appropriate PF function. The program also processes multi-card requests through use of the PP routine, ACE.

ACE
A request can be made through ACE to either read the next control card into RA+7 or to backspace the control card pointer one card (in case an illegal field is detected in a continuation card).

FNT/FST Modifications

RSM 1771



APF Pointer
The presence of a non-zero APF pointer byte for a local (type = 3) file identifies a permanent file.

Permission Bits
These four bits are the security link between the permanent file monitor and the rest of the SCOPE system.

PF Permission Checking (4ES, CIO, OPE, CLO, 1EJ)

RSMs 1808, 1809, 1810

Certain checks of Permanent File permission bits (in FNT) had to be made by SCOPE system routines to prevent security violations.

Read Checks (CIO)
User must have "read" permission for any read operation on a permanent file.

Write Checks (4ES)
If user has both "modify" and "extend" permissions, all write operations are legal.

If user has only "modify" permissions, he may not be positioned at end-of-information when the write request is issued.

If user issues a REWRITE request, he must have "modify" permission.

Open Checks (OPE)
For Open-read, user must have "read" permission. For Open-write, user must have "modify" or "extend" permission. For Open-alter, no permission checks are made.

Close Checks (CLO, 1EJ)
Modifications to CLO apply to processing of random file indexes only.

For a permanent file, writing of the index will be suppressed except when the file is open for "write" or "alter" with "extend" permission. This reduces the incidence of imbedded indexes which can be produced under the current system.

For non-permanent files, the index must be written for Close as well as a Close-unload request (provided the file is open for "write" or "alter"). This is required so Catalog and Extend can verify that the latest index has been written by checking the security field for "close".

End-of-Job Processing (6PC)
1. Multi-read drop: decr. APF entry
2. Non-extend drop: release RBT chain, FNT entry, APF entry
3. Purged drop: release RB's +(2).
4. Extend drop: Catalog vs. CM chain, release extra RBs +(2).

Other Threats to PF integrity (1AJ, CTS, 4ES, DSD)

RSM 1765, 1768, 1819, 1826

Attempt to COMMON--Privacy Violation (1AJ, CTS)
Any attempt to common a permanent file, should it succeed, would permit a user to circumvent the privacy system. A COMMON request via control card or macro is intercepted, a warning message is sent to the dayfile and the request is ignored.

Attempt to Read-release (4ES)
Any attempt to do a read-release function on a permanent file will produce a warning message. The request is ignored.

Attempt to EVICT (4ES)
Any attempt to EVICT a permanent file will produce a warning message. The request is ignored

Attempt to OFF the PF Device (DSD)
The PF device is flagged by a bit set in the EST entry by Deadstart.

DSD intercepts an attempt to OFF this device and outputs a warning message.

18.11 AUDIT

General

The AUDIT utility routine is to provide the installation with basic accounting information. AUDIT produces printed reports containing the following information:

Permanent File Name
Owner
Cycle Number
Creation Date
Expiration Date
Date of last access
Number of attaches
Size in PRUs

AUDIT runs in two modes. One mode produces a printed report of all the permanent files in system, and the other produces a printed report of only those permanent files which have passed their expiration date.

The AUDIT utility consists of a PP routine EPF and of CP routines AUDIT, AUDATE, PICOUT, PARAM, GET, and AUDPP. The CP routines generate an absolute overlay.

The PP routine EPF is responsible for building the CM table for AUDIT containing the first word of each RBR header in the system and for searching the PFD sequentially setting up one at a time an FNT entry for each cycle of every PFD entry. The FNT points to the position of the RBTC entry of that cycle within the ORBTC file. EPF then returns control to the CM program.

The group of CM programs reads the RBTC, calculates the expiration date, and if a selective AUDIT has been requested tests to see if the file has passed its expiration date. If it has not, EPF is called for the next file. If the file has expired or if a complete Audit has been requested, the file size is calculated and the resulting information output. EPF is then called to get the next file and the process continues until the whole PFD has been scanned.

AUDIT uses two files, OUTPUT which may be set to any file name by specifying the file in the program call at execution time (e.g. AUDIT (OUTFILE)) and AUDFILE, a file set up and used internally and which is zeroed at AUDIT completion.

EPF

A. Initial Call (byte 2 of input register is non-zero)

The number of RBRs in the system calculated (FWA

DST table--FWA of RBR table/38} and compared with the available room in RBRWD CM array. If the CM array is not large enough to contain the first word of every RBR header in the system, a message is written telling the amount of space needed and the job aborted. To increase the array size both EPF and the CM routines must be reassembled

```
        Col 6   Col 11
{RBRWDSZ         EQU          50          EPF.82
 RBRWD           BSS          50          CPAUD.15
            2   RBREXP{10},RBRWD{50       AUDIT.10
```

If the CM array is large enough, EPF copies the first word of each of the RBR's into the CM array and then begins the PFD scan. {See C}

B. Subsequent Calls {byte 2 of input register is zero}

A test is made to see if the copy of EPF is from the delay stack. If it is, then EPF checks the incomplete bit in word W.CPPF1 of control point area to see if it is off {byte 1 = 0}. If the incomplete flag is set, EPF checks the error flag to see if AUDIT was aborted or dropped. If the error flag is set, EPF zeros out the FNT for AUDFILE and drops. This mechanism prevents the end of job processor from releasing the RBT chain of ORBTC if AUDIT is aborted or dropped.

If the error flag is not set, a copy of EPF is put into the delay stack for 2 seconds and this copy of EPF drops out. If EPF is called by the CP routine, it goes to get the next cycle in the PFD entry which is being audited. {See C1}.

C. PFD scan

Starting at subdirectory one, the whole PFD is scanned sequentially. The first PFD PRU in the first non-empty subdirectory is read. The first PFD entry in the PRU is scanned for the first non-zero cycle number. {continue at C2}

C1 If all five cycles have been processed, then the next PFD entry is scanned. If all the PFD entries in the PP buffer have been scanned, the next two PRUs are read. This process continues until the whole directory has been scanned. {see D}

C2 The cycle is checked to see if it is either dumped {bit 11 in cycle set} or if the cycle incomplete flag is set {bit 10 in cycle set}. If either flag is set, EPF sets the exit code to 5 or 7, respectively,

and gives control back to the CP program. If the cycle is not dumped or incomplete, then an FNT entry with the file name AUDFILE is set up pointing to the RBTC entry of the cycle within the ORBTC file. The permanent file name, cycle number and RBTC displacement within the PRU are written to CM. A test is made if this is the initial call to EPF, and if it is a copy of EPF it is put into the delay stack for two seconds. The exit code is set to 1 in either case, and EPF returns control to CM program.

D. When the entire PFD has been scanned, the exit code is set to three, the completion flag in byte 0 of word W.CPPF1 of the control point area set to 0 and the FNT, if any, is zeroed out. EPF then drops out.

AUDIT

AUDIT first initializes masks and the page count and then calls PARAM to get the current date and the type of AUDIT to be executed. If the number returned from PARAM is two or greater, a selective AUDIT is to be executed. Next AUDPP is called to issue the initial call to EPF (FLAG=0), to read the AUDFILE file and to position the out pointer in AUDFILE FET to the first word of the RBT chain of the cycle being audited. If the exit code (EXT) from AUDPP is three, the whole directory has been scanned and AUDIT outputs the message "AUDIT FINISHED" and terminates. If the exit code is five, then the cycle was dumped and "THIS CYCLE HAS BEEN DUMPED, NOT AVAILABLE TO SYSTEM" is output in place of the dates, and access information. If the exit code is seven, then "THIS CYCLE IS INCOMPLETE" is output in place of the dates and access information. This message means that while cataloguing this cycle PFM or the load utility was stopped. For this slot in the PFD to be used again, a back-up dump of the permanent files must be taken, an initial deadstart of the system done, and the back-up tape reloaded.

Exit code equal to 1 means that a cycle has been found and the RBT chain is in the CM buffer. The retention period of the cycle is examined. If it is 999 and a full AUDIT is being executed, then in place of the expiration date "INFINITE" is output. If a selective AUDIT is in progress, FLAG is set to 1 and AUDPP called to get the next cycle. In either AUDIT mode if the retention period is less than 999, the actual expiration date is calculated and stored in RMTH, RDAYS, and RYEAR.

If the AUDIT mode is selective and the file has not expired, the cycle is ignored, the FLAG set to 1 and AUDPP

called to get the next cycle. Otherwise the file size in PRUs is calculated. The size is calculated by examining the RBT chains. GET is called to separate the 10 bytes of the RBT word pair into 10 CM words. The starting RB byte, last PRU+1, RBR ordinal and allocation type are gotten from the RBT first word pair and PICKOUT is called to get the device type from the RBR header word in the RBRWD array pointed to by the RBR ordinal. Each RB byte in the word pair is examined and GET called to expand the next word pair. This sequence continues until the whole RBT chain has been searched. The device type and allocation type + 1 are used as indexes into the RBR array to get the number of PRUs/RB for calculating the file size. When an RB change byte is found, PICKOUT is called to reset the device type and allocation type from the RBR header pointed to by the new RBR ordinal. For device types 1 and 4 (6603 and 6603II) if allocation is 0 or 3, the RB byte is examined for inner or outer zone to get the PRUs/RB. After the entire chain has been scanned and the file size calculated, the statistics of the cycle are output, FLAG set to 1 and AUDPP called to get the next cycle.

COMPASS Subroutines Used By AUDIT

1. PARAM

Argument list

IPARAM

Uses the following blank common variables

CURDATE

Uses direct cell

RA+64

Function:

To store into IPARAM the number of parameters passed on to AUDIT through the control card. PARAM gets this information from 0-18 of RA+64. PARAM also uses the macro JDATE to set CURDATE(1) = number of days from the Julian date and CURDATE(3) = year.

2. AUDPP

Argument list

FLAG,EXT

Uses the following blank Common Variables

CREATE	OWNER
DATABUF	PFNAME
DISPLAC	RBRWD
LASTACC	RETEN
NUMBAT	

Uses the following variables in FET named COMMON

AUDFILE BUF

ERROR MESSAGE - AUDIT ABORT - READ PAST EOR ON ORBTC

Function:

Sets up the call to EPF. If FLAG is 0, the call to EPF is the initial call. FLAG equal to 1 flags the subsequent calls to EPF. AUDPP issues the EPF call and then recalls until EPF has completed. AUDPP sets EXT from the exit code returned from EPF. If the exit code is 3 or greater AUDPP exits. Otherwise it clears any EOR bits in the AUDFILE FET, initializes the buffer parameters, and reads the RBTC entry from AUDFILE. AUDPP using the displacement finds the start of the RBTC entry within the buffer and sets the following values; CREATE(1), CREATE(3) = number of days from Julian date, and year of cycle creation, RETEN= number of days the file is to be retained, LASTACC (1,3) = number of days from julian date and year of last ATTACH of this cycle, NUMBAT = number of times this cycle has been attached and OWNER = owner's name left justified blank fill. AUDPP then positions the OUT pointer to the first word of the RBT chain in the RBTC entry and exits. If an end of record is read before the first word of the RBT chain of the cycle was found, the error message - AUDIT ABORT - READ PAST EOR ON ORBTC - is issued and the program aborted.

3. GET

No argument list.

Uses the following blank COMMON variables

RBTEXP

Uses the following variables in FET labeled COMMON

AUDFILE

Function:

GET tests if the next RBT word pair is in the buffer. If it is not a read is issued. When the next RBT word pair is in the buffer, GET expands the 10 bytes into the 10 word array RBTEXP. The OUT pointer is positioned at the next word pair and GET exits.

4. PICKOUT

Argument list

RBRHEAD, the first word of the appropriate RBR Header

Blank Common Used

ALLOC
DEVTYP

Function:

To get from RBR header bits 54-59 the device type and from bits 18-23 the allocation type.

5. AUDATE

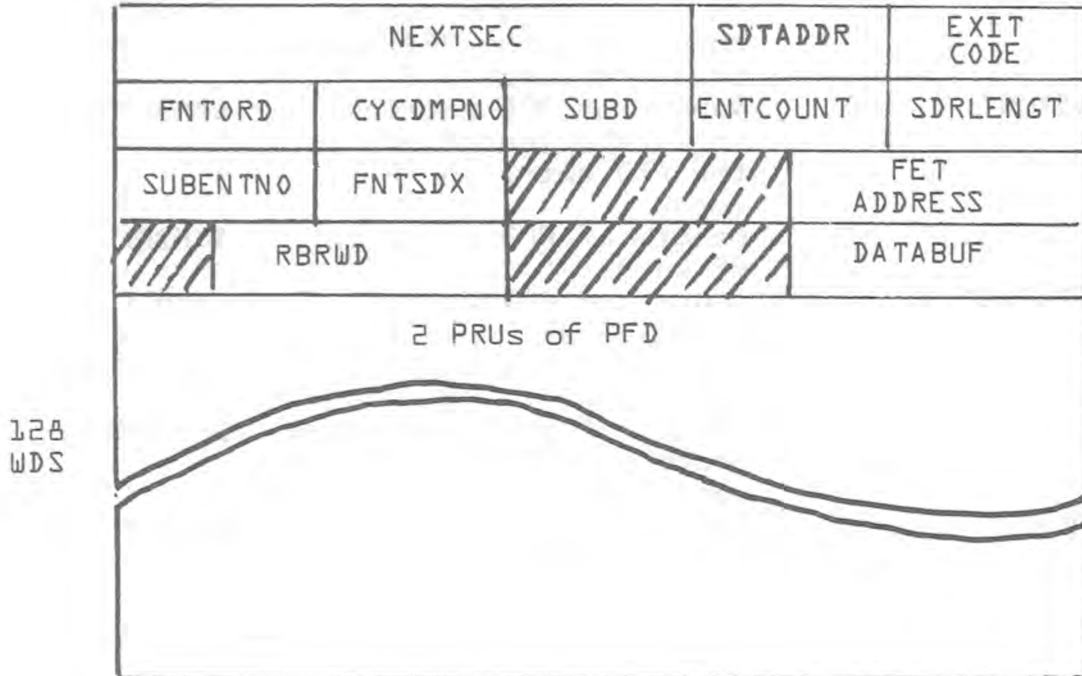
Argument list

IDYCNT, IMTH, IDAY

Function:

To calculate the month and day from the julian date found in IDYCNT and store the results in IMTH and IDAY.

CP, PP Communication Area



Exit Code
- 1 Normal
- 3 Final
- 5 Dumped cycle
- 7 Incomplete Flag Set

BLANK COMMON Used

1 PFNAME {4}
5 CYCLE NUMBER
6 RBTC entry displacement in PFU
7 RETENSION CODE
8 NUMBER OF ATTACHES
9 OWNER
10 device type
11 ALLOCATION type
12 CREATION DATE {3}
15 CURRENT DATE {3}
18 Last access date {3}
21 RBT EXPANSION AREA {10}
31 RBR HEADER WORD ARRAY {50}

FET Common

AUDFILE Fn
FIRST
IN
OUT
LIMIT

Constants in DPF and EPF which must be changed if changes are made to PFM, PFD or RBTC entry sizes

Symbol	Value	Meaning	id.*
CYCLPOS	25	Position of the cycle pointer in the PFD header in PP words + 5	SYMDEF.37
LASTENT	320	Number of PFD entries in a PRU* length of the PFD entry in PP words	SYMDEF.40
PFDLTH	80	Length of PFD entry in PP words	SYMDEF.46
PFDLTHCM	16	Length of PFD entry in CM words	SYMDEF.47
PFDNO	3	Number of PFD entries per pru-1	SYMDEF.48
SETRBTC	60	Length of the fixed size header of the RBTC in PP words	SYMDEF.53
TPOS	2	Byte position of T in word of RBTC	SYMDEF.55

System Dependent Variables

RBRWD	50	Array contains the first word of every RBR header in the system

Statement Format

```
Col 1       Col 6            Col 11

        2 RBTEXP(10, RBRWD(50)               AUDIT.10
RBRWD                     BSS 50
```

The array RBR (64, 65) contains the number of pru's /RB for each allocation type and device type where Row = device type and Column = Allocation type +1. There is defined in the release program the values for the following device types and allocation styles

Device Type	Allocation Style 00	01	02	03	10
01	50/64	50	64	50/64	8
02	50	50	-	50	8
04	50/64	50	64	50/64	8
07	5	5	-	5	-
11B	21	-	-	21	-
12B	21	-	-	21	-

Any modification of these values or addition of device types and/or allocation styles will require modification to the array and reassembly of AUDIT.

Program Name	Symbol	Page
EPF	X	1
AUD1	AA1	4
OUT	AA2	3
EPP11	AA3	3
AUDPFD	AA5	4
SUDSCAN2	AA6	2
AUDIT	X	7
2000	AA7	7
3100	AA8	9

Routines Documented	Page
EPF	1
INIT	1
DLYSTIC	1

ERROR MESSAGES

EPF ABORT - NO RBTC ENTRY IN FNT
The AUDIT routine was called into a system which does not have the Permanent File System RBTC file defined.

EPF ABORT - NO ENTRY IN FNT FOR OSD000
The Audit routine was called into a system which does not have the permanent file system defined.

WAITING FOR FNT SPACE
{B-Display only}
{Job can be dropped by operator}

EPF ABORT - NO. OF RBRS EXCEEDS RBRWD BY XX
The number of RBR entries in the system exceeds the CM array size in Audit by XX words. The CM array must be large enough to contain the first RBR header word for each RBR entry.

EPF ABORT - SYSTEM ERROR NO. XX
PF has found an Audit
XX = 1 Bad Address for CM write

AUDIT ABORT - READ PAST EOR ON ORBTC
Audit has read an end-of-record on ORBTC before the RBTC entry was complete.

18.12 DPF - DUMPF

General

The dump utility provides for the user the capability of dumping permanent files to tape. The dump program operates in two modes. One gives the installation the ability to take periodic back-up dumps of all the permanent information in the system, and the second permits the installation to down a unit by clearing it of all permanent information.

Two routines comprise the dump utility. One is a PP routine named DPF and the second is a CP routine called DUMPF. The function of DPF is to read and manipulate the information in the PFD and RBTC.

DPF reads the PFD sequentially searching for an entry to dump. When such an entry is found, PDF reads in the RBTC entry for that cycle. For a UNIT dump DPF scans the chains for any reference to the specified unit. If a reference to the specified unit is found or if the dump is a back-up dump, DPF chains the RBTs to high core, sets up an APF table entry, and a FNT entry for the file, and then drops out.

DUMPF reads the file that has been set up by DPF and copies it to tape. When end of information on the file is reached, DUMPF issues a call to DPF to get the next file. This procedure is followed until the entire PFD has been searched.

DPF

Upon entry, DPF does initialization and then, unless DPF was returned from the delay stack, verifies that it was called to a control point which has a copy of the permanent file DUM attached to it. If it was not, a message is issued and the job aborted.

After verification, DPF determines whether the PFD PRU to be processed is to be read from the disc or copied into the PP buffer from the CM buffer. If bit 3 of the exit/entry information in the status word is set, then a copy of the PFD PRU is in core. When the PFD is to be read from CM, DPF also checks if a unit dump is in progress. For a unit dump a further check is made to see if it is necessary to set the entry-free flag in the RBTC and rewrite the PFD and RBTC entries.

Types of Calls

A. Initial Call (Entry code = 0)
   1. DPF checks to see if any other dump programs are running. If not, go to (A4).

2. If there are, DPF checks the dump mode. If the dump is a unit dump, DPF checks if the PFD and RBTC or the dump permanent file is on the specified unit. If either of the cases exist, a back-up dump is forced.
3. DPF puts the type of dump in the label area of the dump tape FET and begins to search for files to dump. (See B.)
4. If there are no other copies of DUMPF running, DPF checks the APF table to see if the Permanent File Manager (PFM) is active. Upon finding PFM activity, DPF puts itself into the delay stack and drops out. This process continues until PFM activity has ceased.

   When the PFM system is quiet, DPF reads the entire PFD and clears the dump bit in each PFD entry header, sets the utility flag, and then continues at (A2).

B. Normal File Processing (Entry code = 6 or 7)
1. If the dump utility is in the middle of dumping the cycles of a PFD entry, go to (B6) to get the next cycle.
2. The total number of entries in a subdirectory is checked against the number of entries already processed. If the limit of the subdirectory has not been reached continue at (B3). If it has, the next subdirectory is checked. This procedure continues until all of the subdirectories have been checked.
3. The next entry in the PFD sector which was read into the PP buffer, either from disc or from the CM array is processed. If all the PFD entries in that PRU have been processed, the next PRU in the subdirectory is read.
4. A check is made that there is no entry in the APF table with the same PFD pointer. If there is not, go to (B5). If there is already an APF entry, the type of dump type which has it attached is compared with the dump type of this copy of DPF. If both are full dumps or both unit dump of the same unit, the PRU is ignored by this copy of dump, the number of entries processed incremented, the next PRU read and continue at (B). If this dump type is a unit dump and there is a full dump running, then this copy of the dump program is stopped and a message issued. In the other cases the PFD PRU pointer and number of entries in that sector are saved in a CM array to be tried again at a later time. Continue at (B2).
5. The dump flag in the PFD header is checked and if it is set processing, continues at (B2).
6. If not, the cycles are scanned until one which has not been dumped is found.
7. The PFD header information is dumped to the DUMP tape buffer.

8. If the type of dump is a back-up dump continue at (B9). If it is a unit dump, the RBT chain in the RBTC entry for the cycle is read and scanned for any reference to the specified unit. If none is found, processing continues at (B6). If a reference is found, the address of the RBTC entry is saved for rewritting the RBTC with the entry free flag set after the file has been copied.
9. The RBT chain in the RBTC is copied into high core. A FNT for the file is created, and the RBTC header is written into the dump tape tape buffer. If a back-up dump is being performed, the RBT word pairs of the previous file, if any, are cleared. In both types of dump the RBT chain is then released to the empty chain. This accomplishes for a back-up dump the RBT chains being released but not the RBs, and for a unit dump both the RBT chains and the RBs being released.
10. DPF saves crutial variables in W.CPPF1 and W.CPPF2 and the status word pair in CM. Copies the PFD entry and addresses of next sector of PFD to CM buffer and drops.

C. Processing PFD sectors on which multiple copies of the dump program conflicted. (Entry code 2 or 3)
  1. If the dump utility is in the middle of dumping the cycles of a rescanned PFD entry proceed as in (B6) - (B9) except that if there is a matching APF entry, the return is to (C4).
  2. If the PFD sector is in the CM buffer, read it into the PP Buffer and go to (C6).
  3. Save the pointers in W.CPPF1 and W.CPPF2 in CM array.
  4. If all PFD PRU entries have been rechecked, go to (C9).
  5. Read in the address of the next PFD PRU to be re-checked, save the number of PFD entries in the PRU, and set the subdirectory number and FNT address for that subdirectory. Set the exit code to 30B.
  6. Processing the PFD sector proceeds as in (B4) - (B9) except that if there is a matching PAF entry, the return is to (C4).
  7. If all PFD entries in the PRU have been examined, go to (C4).
  8. Otherwise processing of the PFD entires continues on the PFD sector in the PP Buffer at (C6).
  9. Restore the W.CPPF1 and W.CPPF2 from CM array and set exit code to 60B, save the critical values and drop.

D. Termination (Entry code = 4)

1. All processing has been completed or DPF aborted or stopped. If a back-up dump had been performed go to {D3}.
2. If a unit dump was in process, the APF pointer in the FNT is set to zero if an FNT had been created for XXXTENP. This process will cause the end-of-job processor to release the RBs of the last file copied.
3. The number of control points attached to the dump program is checked. If other control points are still attached, go to {D4}. If this is the last running copy, then the utility flag is cleared.
4. For normal termination set Exit code to 120B, and drop the PP. If DPF stopped, set exit code to 130B and drop the PP.

E. Parity Error {Entry code = 9}
Output message and set abnormal exit flag and go to {D1}.

F. IO Error {Entry code = 8}
An IO error occured in CM program. Flag an abort and go to {D1}.

DUMPF

DUMPF uses two files, the tape file PERMDMP and the disc file XXXTEMP. The XXXTEMP file is set up by DPF and is flagged as a permanent file. For a unit dump the APF flag is zeroed at job completion. The tape file is rewound prior to writing and unloaded at job termination.

A. DUMPF requests a standard SCOPE labeled tape at 800BPI density. If the request is satisfied DUMPF continues, otherwise an error message is issued. DUMPF then determines whether the dump mode is to be a back-up or unit, by checking whether a parameter was passed to it by the control card. If the dump is a unit dump, the parameter is converted to an octal number and placed in bits 12-23 of the first status word for CP-PP communication. {See status word pair diagram.}

B. DUMPF then requests 10,000B word of CM, and moves the file header information to the output tape buffer.

C. DUMPF then calls the PP routine, DPF. When control is returned to the CP program, the status is checked.
   C1. If the exit code equals 13XB close unload the dump tape and terminate without message. The PP routine was stopped and has already issued the explaining message.

C2. Exit code equal to 12XB. Close unload the dump tape and output ▽DUMPF FINISHED▽ and end.

C3. Exit code equal to 2XB. A 2XB flags that DPF is to rescan the PFD entries in the CM array. The status word one is set to 0,0, last entry stored +1, EST ordinal, entry, continue at C.

C4. Exit code equal to 6XB. This exit code flags that a rescan of all the PFD entries has been completed. If the rescan was initiated because the whole PFD had been scanned and the entries in the CM array were the only remaining undumped entries (FINAL=1) then if no more conflicts were found (STORORD=0) then entry code is set to 40B and continue at C. If there are still entries which have not been processed (STORORD≠0) the DUMPF goes into timed RECALL, sets the entry code to 20B and continues at C3. If the rescan was initiated because the CM array had been filled (FINAL=0), a check is made to verify that the array is not still full (MAXORD = STORORD). If the array is full, DUMPF goes into timed RECALL, sets entry code to 20B and continues at C3. If the CM array is not full, set status word one to 0,STORORD,TRUMAX,EST ordinal, entry code and go to C.

C5. Exit code equal to 4XB. Set the flag for final processing (FINAL set to 1) and go to C. This code indicates that the whole directory has been scanned and that only conflicting PFD entries, if any, remain to be checked.

C6. Exit code equal to anything else means that a file has been found to dump.

D. The permanent file name is saved in case of disc parity error, the dump tape file opened if this is the first file to be dumped and the header record output.

The file is now copied to tape, logical record by logical record. When an end-of-information is read, DUMPF writes two ends of file and resets buffer pointers. Processing continues at B.

E. If an IO error occurs, the own code routine ERCHECK is executed. Any error other than a parity error causes DUMPF to output the error code in a message, set entry code to 100B, and go to C.

For a disc parity error the permanent file name is output with the parity error message and copying of the file continues.

For a tape parity error the tape is positioned at the end of the previous permanent file, and an end of file written. The entry code is set to 110B and processing goes to C.

DUMP UTILITY ERROR MESSAGES

1. DPF ABORT - NO ENTRY IN FNT FOR OSD000 {DPF}
   The dump routine was called into a system which does not have the permanent file system defined.

2. DPF ABORT - NO RBTC ENTRY IN FNT {DPF}
   The dump routine was called into a system which does not have the Permanent File System RBTC file defined.

3. DPF ABORT - SYSTEM ERROR NO. XX {DPF}
   DPF has found a dump or PFM system error
   When XX = 1 Bad address for CM write
   = 2 APF pointer does not point within PFD chain
   = 4 CM write beyond LIMIT of tape buffer
   = 5 PFD pointer does not point to PFD header word
   = 6 RBTC pointer in PFD does not point to correct RBTC entry

4. DPF ABORT - UNIT SPECIFIED NON-ALLOCATABLE {DPF}
   The EST ordinal specified in the control card of a unit dump points to a non-allocatable device. Permanent files are only on allocatable devices.

5. DPF ABORT - UNIT SPECIFIED = ECS {DPF}
   The EST ordinal specified in the control card of a unit dump points to ECS. Permanent Files cannot be assigned to ECS.

6. DPF NOT CALLED BY DUMP ROUTINE {DPF}
   The PP program DPF was called to a control point which did not have the dump permanent file, DUM, attached.

7. DPF STOPPED BY SYSTEM {DPF}
The error flag in the control point area was found on.

8. DPF STOPPED - FULL AND UNIT DUMPS RUNNING TOGETHER {DPF}
A full dump has been found running and conflicting with a copy of a unit dump. The unit dump is terminated.

9. DPF STOPPED - PARITY ERROR ON DUMP TAPE {DPF}
A parity error was found on the dump tape and the tape has been positioned back to the end of the previous file dumped. DPF is called to terminate the dump program.

10. DUMPF ABORT - IO ERROR RETURN {DUMPF}
IO error code other than a parity error causes the dump program to abort.

11. DUMPF ABORT - REQUEST ERROR - XX
The request was not completed successfully. XX gives the number of the type or error.

12. DUMPF FINISHED {DUMPF}
The dump utility has terminated normally.

13. DUMPF - PARITY ERROR ON DISC PF NAME -XX... {DUMPF}
A disc parity error was found while coping permanent file XX.... The message is output and coping continues.

14. DUMPF - PARITY ERR WHILE IN OWN CODE {DUMPF}
In processing one parity error the dump program got another parity error while trying to reposition the tape back to the previous file.

15. DUMPING XXX... {DUMPF}
B-Display message only.
XXX... is the permanent file name of the file currently being dumped.

16. RESCANNING SKIPPED PFD SECTORS {DUMPF}
B-Display message only.
DUMPF is rescanning the PFD entries which had to be bypassed earlier because of some conflict. This rescan takes place either when the CM array becomes full or the entire directory has been scanned leaving only the skipped PFD entries to be dumped.

17. WAITING FOR APF TABLE SPACE {DPF}
The APF table in CM is full and the dump routine is waiting for a file to be "DETACHED" to free some space.
{Job can be dropped by operator}
{B-Display only}

18. WAITING FOR FNT SPACE {DPF}
{B-Display only}
{Job can be dropped by operator}

19. WAITING PFM IDLE {DPF}
{B-Display message only}
DPF is waiting for all Permanent File Manager activity to cease before it will begin executing.

Size/Position Defining Variables

Constants in DPF and EPF which must be changed if changes are made to PFM, PFD or RBTC entry sizes.

Symbol	Value	Meaning	ID#
CYCLPOS	25	Position of the cycle pointer in the PFD header in PP words +5.	SYMDEF.37
LASTENT	320	Number of PFD entries in a PFU* length of the PFD entry in PP words.	SYMDEF.40
PFDLTH	80	Length of PFD entry in PP words.	SYMDEF.46
PFDLTHCM	16	Length of PFD entry in CP words.	SYMDEF.47
PFDNO	3	Number of PFD entries per pru-1.	SYMDEF.48
SETRBTC	60	Length of the fixed size header of the RBTC in PP words.	SYMDEF.53
TPOS	2	Byte position of T in word of RBTC.	SYMDEF.55

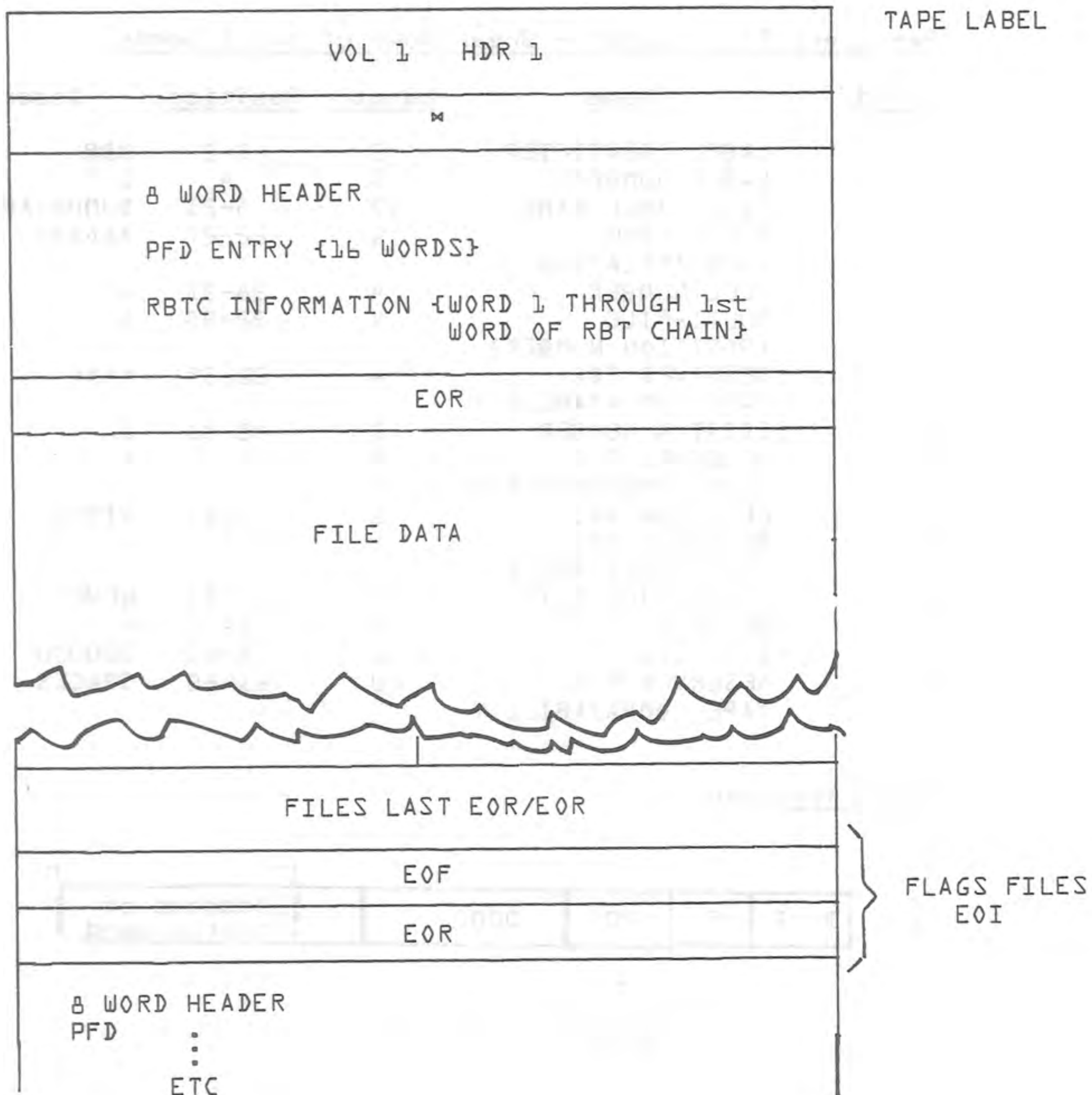
In DUMPF the variable determining the number of PFD PRUs to be saved before causing a rescan of the list is TRUMAX

TRUMAX 10 SCP316K.320

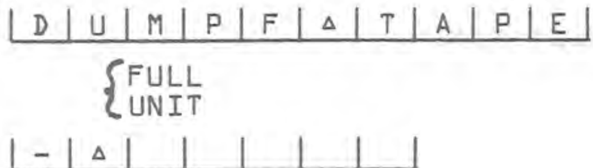
TAPE FORMAT

Multi-Reel File

--



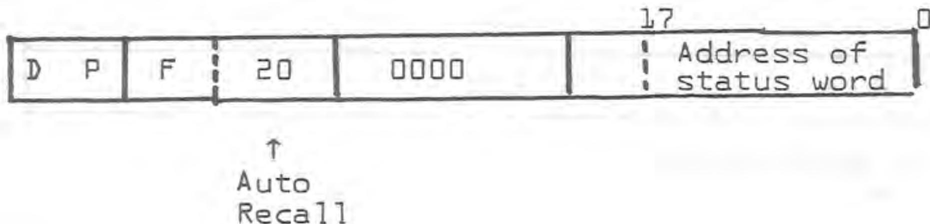
Information in the 17 Charachters of the Label



Permanent File Header - Break Down of the 8 Words

Field	Name	Length	Position	Description
1	LABEL IDENTIFIER	3	1-3	HDR
2	LABEL NUMBER	1	4	1
3	FILE LABEL NAME	17	5-21	DUMPΔTAPEΔOFΔP.F.
4	MULTI-FILE IDENTIFICATION	6	22-27	ΔΔΔΔΔΔ
5	REEL NUMBER	4	28-31	1
6	MULTI-FILE {POSITION NUMBER}	4	32-35	1
7	RESERVED FOR TAPE COMPATABILITY	4	36-39	ΔΔΔΔ
8	EDITION NUMBER	2	40-41	1
9	RESERVED FOR TAPE COMPATABILITY	1	42	Δ
10	CREATION DATE	5	43-47	BIRTH
11	RESERVED FOR TAPE COMPATABILITY	1	48	Δ
12	EXPIRATION DATE	5	49-53	NEVER
13	SECURITY	1	54	X
14	BLOCKCOUNT	6	55-60	000000
15	RESERVED FOR TAPE COMPATABILITY	20	61-80	SPACES

INPUT Register



HEADER INFORMATION FOR EACH FILE DUMPED

8 WORD HEADER-FORMAT THE SAME AS LABEL HEADER ON TAPE

H	B	R	1	D	U	M	P	Δ	T
A	P	E	Δ	Ø	F	Δ	P	.	F
.	Δ	Δ	Δ	Δ	Δ	Δ	0	0	0
1	0	0	0	1	Δ	Δ	Δ	Δ	0
1	Δ	B	I	R	T	H	Δ	N	E
V	E	R	X	0	0	0	0	0	0
Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ
Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ	Δ

PFD ENTRY OF DUMPED FILE

7 7 7 7 7 7 7 7	P	F	D Δ	FLAGS
PERMANENT FILE NAME UP TO 40 CHARACTERS				
CYCLE NUMBER	CYCLE NUMBER	CYCLE NUMBER	CYCLE NUMBER	CYCLE NUMBER
RBTC POINTER	RBTC POINTER	RBTC POINTER	RBTC POINTER	RBTC POINTER
PASSWORDS: TURN-KEY, CONTROL, MODIFY, EXTEND, READ				
UNUSED BUT RESERVED				

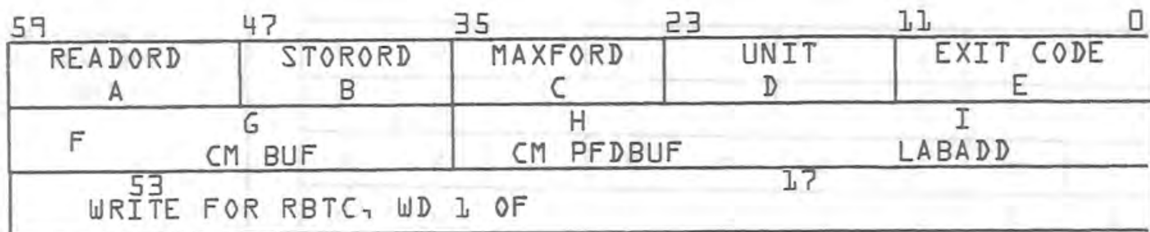
RBTC ENTRY UP TO AND INCLUDING THE 1ST WORD OF THE RBT CHAIN

7 7 7 7 7 7 7 7	R	B T	C	FLAGS
PERMANENT FILE NAME UP TO 40 CHARACTERS				
POINTER TO PFD ENTRY			EST ORDINAL	CYCLE NUMBER
CREATION DATE			RETENTION PERIOD	
DATE OF LAST ACCESS				
RESERVED				
NUMBER OF ATTACHES	RESERVED	RESERVED	W	Y
OWNER ID				
0	POINTER TO S	POINTER TO T	SUB-DIRECTORY NUMBER	
VARIABLE LENGTH SLOT RESERVED BY CDC				
S: INSTALLATION RESERVED SLOT				

1ST WORD OF RBT CHAIN

T: NEXT WORD PAIR	RBR ORDINAL	1ST RB BYTE		ALLOC TYPE	LAST PRU+1	ETC

Status Words

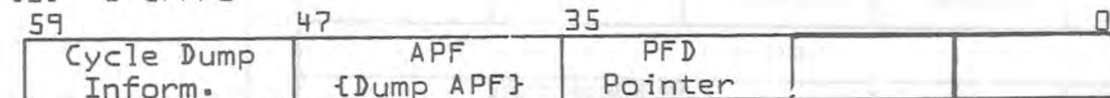


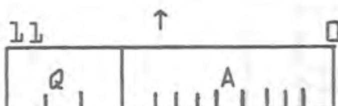
A = ordinal of the next entry in the CM array to be re-checked
B = ordinal of thenext space in the CM array to fill
C = maximum number of skipped entries to be saved prior to reviewing the list
D = EST ordinal if UNIT DUMP in bits 11-1 bit 0=0
    0 if FULL DUMP in bits 11-1 bit 0=1
E = Exit information from the PP
    bits 11-3 = 0 Initial call
    = 1 Not used
    = 2 Use CM Buffer for PFD address
    = 3 Get next cycle from Rechecked PFD entry
    = 4 Final call
    = 5 Not used
    = 6 Get next PFD entry from directory
    = 7 Get next cycle of PFD entry
    = 8 IO error, cleanup, and abort
    = 9 Parity error
    =10 End Run
    =11 Abnormal stop but not abort
    bits 2-0
    = 1 Sequential file; and restart CP

F = difference between the INPUT file FET address and the OUTPUT file FET address
G = address of Buffer for skipped PFD entries
H = address of 65 word buffer which holds the pointer to the next sector in the PFD and the PFD sector currently being processed.
I = address of the OUTPUT file FET

Use of CP Words

(1) W.CPPF1





Q = The byte position of the last cycle pointer which was checked for dumping
A = 0; nothing dumped from this PFD entry
= non-zero; cycles were dumped and PFD entry must be re-written

(2) W.CPPF2

FNTSDX	SUBD *	SUBENTNO	FNTORD	APFORD

FNTSDX = Address of OSDX FNT
SUBENTNO = Entry no. in SUBD for next entry
FNTORD = FNT address for XXXTEMP file
APFORD = XXXTEMP APF ordinal

Use of APF entry

59	23	11 6 5	0
PFD Pointer	Unit	Flags	Count

Unit =	bits 11-1	bit 0	Type
	EST ordinal	0	selective dump
	0	1	Full Dump

Subprograms Not Appearing in Flow Chart

CKCPFNT	Searches for a file local to the control point.
DROPABNL	Called to process an abnormal termination but not an abort.
DUMPHDR	Writes the RBTC header information to the tape file buffer.
FINDPFD	Reads the PFD entry, locates the entry within the PRU and checks PFD header word.
PFDFETC	Reads PFD entry pointed to by the APF pointer and checks name for the dump permanent file DUM. Exit with non-zero flags no match. If DUM is found, the dump flag is set in the APF entry and then exits with zero in A register.
PUTLAB	Writes the dump type to the label area of the tape FET in CM.
SETPOS	Reads the RBTC and dumps header information until the first word of the RBT chain is in the buffer.
RSDT	Interlock the SDT table.
DSDT	Drop SDT table interlock.

18.13 LOADPF/LPF

General Description

This utility provides the user with the capability to reload permanent files that have been previously dumped to tape. This routine will reside on the system library and operate at a control point. It will restore all permanent files from a Dump Tape assigned to its control point and regenrate the Permanent File Directory and the RBT Catalogue information. There are two modes of operation: the first being the reloading of a back-up tape, i.e. all the permanent files; and the second, the reloading of files previously dumped from a specific unit. These tasks will be performed by a CP program called LOADPF and a PP program called LPF. LOADPF will handle the input/output requests, whereas LPF will set up the directory and catalogue information.

Function

Initially, LOADPF requests the dump tape, verifies the tape label, and sets the Partial/Full indicator. Then LOADPF reads the file header. LPF is then requested.

LPF's initialization consists of the following checks: Searches the APF table to see whether a copy of LOADPF is all ready in operation and increments the count of load programs operating by one. If there is no load program running, a check is made to verify that there are no files active in the APF table. If there are files active, then LP enters the delay stack. When it returns from the delay stack, it checks the APF table again. This continues until the system is "quiet". Then an entry for the load program is created in the APF table. A dummy FNT entry (DUMFNT) is created for use in copying files.

After initialization, LPF will take the following path for the first and subsequent files on the dump tape attached to this control point. Whenever LPF is entered, a check is made for "Operator Drop" action, and then aborts if so requested. Otherwise, LPF continues its functions.

For a partial dump reload, the file must meet the following criteria to be reloaded:

1. File name in the Permanent File Directory.
2. Cycle with dump indicator set in PFD.
3. Passwords must be the same.

For a full dump reload, the file must meet the following criteria:

1. File name not in the Permanent File Directory, or
2. File name in Permanent File Directory, and
    a. Cycle not in PFD
    b. Passwords must be the same
    c. There must be room for another cycle.

If any of the above conditions are not met, the file is not reloaded.

If the file is new to the system, an attempt is made to catalogue it in its original subdirectory provided that subdirectory exists and is not full. Otherwise, it will be catalogued in the least full subdirectory. The file (pointers to the Permanent File directory PRU) will then be entered into the APF table to ensure that no other load program will affect this file until the current processing is finished.

An attempt will be made to reload a file to its original device. If that is not possible, the file will be loaded to another device with the same allocation type. If there is no device with the original allocation type, the file will not be reloaded. The allocation and EST ordinal are then put in the dummy FNT. The exit code is set to zero. The PP is then dropped.

LOADPF checks the exit code to determine what action to take. If the indication (3) is to skip the file, the tape is read until the next file header (or end-of-information) is encountered. The copying of a file will continue until a double end of file followed by a new file header (or end-of-information) is encountered. After the file is copied, it will be closed but not unloaded. LPF will then be called to create the RBT catalogue.

For a partial dump, LPF will attempt to use the same RBTC space unless this cycle's catalogue is too large for the original entry. In that case, the RBTC will be appended to the end of the RBT catalogue and location in CMR pointing to the next available catalogue space will be updated. On a full reload, the current RBTC will always be appended to the end of the catalgue and the pointers in CMR will be updated. After the RBTC is written out, the RBT word pair chain for that file will be released.

For subsequent files, the procedure will be the same, except that the initialization in LOADPF and LPF will not be necessary.

After the last file is copied and closed, LOADPF will call LPF to decrease the number of load programs operating at this time and clear the FNT.

LOADPF will rewind and unload the tape plus issue a completion message "LOADPF FINISHED".

<u>Subroutines</u>

<u>LPF</u>

ADDRAN	Routine to calculate a CM address. D.RA is added and the 18-bit result is left in the accumulator.
ALLOCRB	Routine to check the allocation type for a specified device.

APFENT	Routine to clear the APF file entry.
BACKSP	Routine to backspace one PRU of the RBT catalogue.
BIND	Routine to convert a binary number to three display characters.
CLEANUP	Routine to decrement the number of load programs operating at this time. If there are no more load programs operating, a completion message is issued and the utility bit in the permanent file toggle byte is cleared.
CORBT	Routine to count the number of words needed for the RBT chain for the current file.
CYCLIC	Routine to search the whole permanent file directory for a given file.
DAPF	Routine to drop the APF interlock bit.
DELAY	Routine that puts LPF in the delay stack for one second.
DIV5	Routine that performs division by 5.
DRBTC	Routine to drop the RBTC interlock bit.
DSD	Routine to drop the subdirectory interlock bit.
DSDT	Routine to drop the interlock bit for thw hole permanent file directory.
ESTCHK	Routine to find an EST ordinal with a given allocation type. A check is made to insure that the device is on.
FDCYC	Routine to find a particular cycle position in the PFD.
FINDRBR	Routine to find start of an RBR table, given the RBR ordinal.
FNTDM	Routine to set up a dummy FNT.
INAPF	Routine to scan APF table for matching PFD pointer.
JADDR	Routine to get address from input register and add D.RA.
JPNCK	Routine to compare the file name of the current file to a file name already in the PFD.
JPSWD	Routine to compare the passwords of the current file to the passwords of a file by the same name in the PFD.
JRSTR	Routine that restores communication values to the PP from the CP.
JSTR	Routine that stores communication values from the PP to the CP.
JTPHR	Routine that copies the file header information from the CP to the PP.
LFLAG	Routine that loads the flag byte of the current APF entry into the accumulator.

LSDB	Routine that loads the subdirectory byte into the accumulator.
NEWAPF	Routine that creates an APF entry for the file PFD pointer.
PAUSE	Routine to create a pause to wait for storage allocation.
PFDIO	Routine so set up a stack request for either a read or a write operation.
PFNDF	Routine to output the file name and cycle number.
PRAPF	Routine that uses PAUSE when there is no APF space available.
READPFD	Routine to read one PRU of the PFD.
READRBT	Routine to read one PRU of the RBTC.
REND	Routine to write the permanent file toggle byte to CM.
RRBTC	Routine to set the RBTC interlock bit.
RS	Routine to request the permanent file toggle byte.
RSD	Routine to set the interlock bit for a particular subdirectory.
RSDT	Routine to set the interlock bit for the whole subdirectory.
RSDB	Routine to write the subdirectory byte to CM.
RWCOM	Routine to set up PP tables for either reading from CM or writing to CM.
SDSRC	Routine to search FNT for a given subdirectory.
SEARCH	Routine to search a given subdirectory for a file.
SECNT	Routine to update the subdirectory count and check for 4/5 full or completely full.
SELSUB	Routine to select the least full subdirectory.
SRCHFNT	Routine to search the FNT for a given file.
SRPRV	Routine to search one PRU of the PFD for either a given file name or an empty slot.
TABSV	Routine to save tables - PFN and JFILE.
WRTPFD	Routine to write one PRU of the PFD.
WRTRBT	Routine to write one PRU of the RBTC.

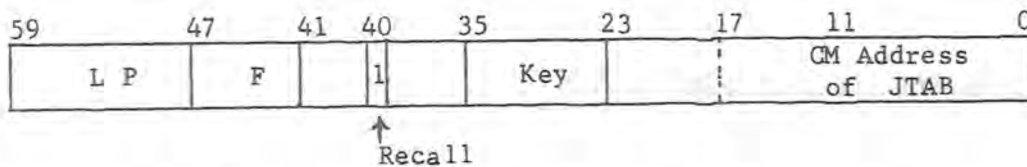
LOADPF

BUFOR	Routine used to insure efficient Read or Write.
CLOFILE	Routine to close a file and call LPF to create the RBTC.
ERCHECK	Owncode routine used when copying data from tape to unit. A parity error will cause an error message and then copying will continue. A system error will cause the load program to abort.

ERROR	Routine to format the error message and check for type of error.
GETCY	Routine to get the cycle number from the file header and convert it to display code.
HEDERR	Owncode routine used when reading the file header from tape into the CP. The file will be skipped if a parity error occurs but the load program will be aborted if a system error occurs.
PAREX	Routine that calls the PP program LPF.
READREC	Routine that sets up call to BUFOR to copy one record.
SYSTEM	Routine used to place requests to Monitor for PP action in cell RA+1.

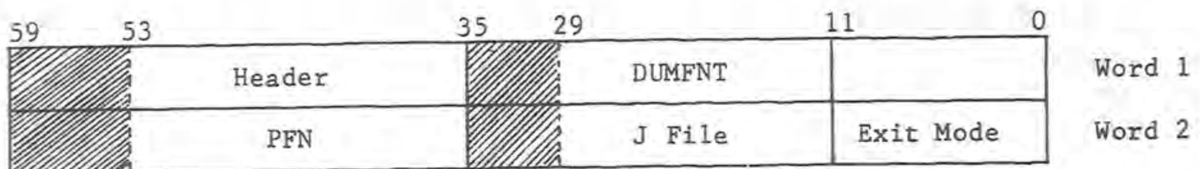
Communication

Input register - communication from LOADPF to LPF



Key (bits 24-35)
0 = New file but not first time
1 = Close file - create RBT catalogue
2 = Final pass
3 = First pass - partial dump
4 = First pass - full dump

CM address of JTAB (bits 0-17) communication table with the following format:

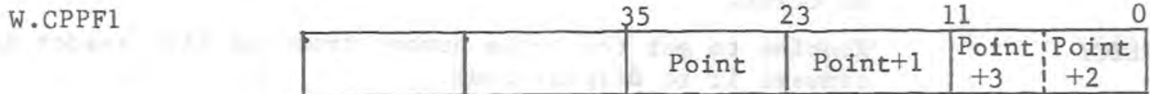


Word 1 bits 36-53 CM Location of file header
bits 12-29 CM Location of FET used in copying information

Word 2 bits 36-53 CM Location of current permanent file name
bits 12-29 CM Location of communication table
bits 0-11 Key set by LPF and used by LOADPF
0 - indicates that the file should be copied
3 - error condition; do not copy file

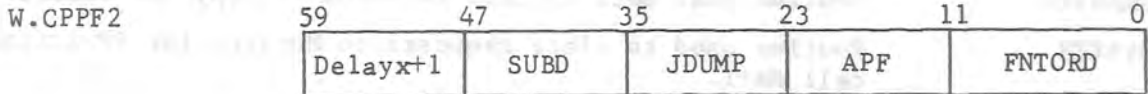
Control point saved words:

W.CPPF1



PFD Pointer: Point - RBT address
Point+1 - RBT ordinal
Point+2 - PRU number
Point+3 - Displacement in PRU

W.CPPF2



DELAYX+1 - return address from delay stack
SUBD - subdirectory number
JDUMP - 0 = partial dump, 1 = full dump
APF - APF ordinal
FNTORD - FNT ordinal

PPRES - Routines that are used by LPF

R.DCH
R.DFM
R.EREQS
R.IDLE
R.MTR
R.PAUSE
R.RCH
R.READP
R.TFL
R.WRITEP

System Macros Used

PPENTRY
LDK
UJK

Control Point words used

W.CPPF1
W.CPPF2

CMR words used

P.EST
P.FNT
P.PFM1
P.PFM2
P.RBR
P.RBT
P.RQS

CMR word changed

P.PFM2

SCPTEXT symbols used

CH.FNT	C.RBRUNT	C.STPPRU	L.CPNUM
CH.PFM	C.RBTC1	C.STPRBA	M.ABORT
C.APF	C.RBTC2	C.STPRBN	M.DPP
C.APFL	C.RBTC3	D.CPAD	M.RPJ
C.CPEF	C.RBTWPL	D.FA	O.BPRU
C.CPFL	C.SDL	D.FL	O.RCHN
C.CPNUM	C.SDT	D.FNT	O.RDP
C.CPRA	C.SDTL	D.PPIR	O.WRP
C.FFRBA	C.STCPU	D.PPIRB	
C.PFMCH	C.STFB	D.PPMES1	
C.PPFWA	C.STO	D.PPONE	
C.RBRA	C.STPFW	D.RA	
C.RBRAD	C.STPLW	LE.FNT	

LOADPF - Uses of CPC:

CLOSE AND CLOSE UNLOAD
MEMORY
MESSAGE
OPEN
RECALL
READ
READSKP
WRITE
WRITEF
WRITER

MESSAGES

CYCLE XX, NNN...N
XX is the cycle number, NNN...N is the permanent file name.

DIRECTORY ALMOST FULL
Permanent file directory is over 80% full.

FD-ALL CYCLES FULL
Full Reload, no spare cycles are available for this file - file is skipped.

FD-CYCLE ALREADY IN PFD
Full Reload, cycle already exists in the system - file is skipped.

FD-FILE ASSIGNED TO ANOTHER DEVICE
Full Reload, allocation type is not available on original device.

FILE IN SUBDIRECTORY XXX
    XXX refers to the subdirectory number.

INCORRECT DUMP TAPE MOUNTED
    Mount correct dump tape.

IS DUMP TAPE - FULL (UNIT)
    Type GO with control point preceding.

LOADPF ABORTED - SYSTEM ERROR XX
    XX refers to error code

LOADPF FINISHED

LPF ABORT - BAD ADDRESS
    Address out of range (outside field length).

LPF ABORT - NO ENTRY IN FNT FOR OXXXX
    XXXX refers to RBTC or subdirectory.

LPF ABORT - NO RBTC SPACE
    The RBT catalogue is full.

LPF FINDS NO FNT SPACE
    Waits for space.

LPF - FINISHED LOADING
    This is the last copy of reload running.

LPF STOPPED BY SYSTEM
    Operator drop.

NO EOR, EOF, OR EOI
    Load aborted.

NO EQUIPMENT AVAILABLE WITH ALLOCATION TYPE FOR
    File is skipped.

PARITY IN HEADER, WILL NOT RELOAD FILE

PARITY OCCURRED IN RELOADING

PD-CANNOT FIND FILE
    Partial reload - file name not in system, file is skipped.

PD-CYCLE NOT DUMPED
    Partial reload - file skipped.

PD-CYCLE NOT IN PFD
    Partial reload - file skipped.

PD-FILES ASSIGNED TO ANOTHER DEVICE
    Partial reload - allocation type not available on original device.

PF NAMES COMPARE BUT NOT PASSWORDS
    File is skipped.

PFD FULL - CANNOT RELOAD FILES STARTING WITH
CYCLE XX,NNN...N
AND FOLLOWING

PFN FOUND IN SD XXX
    XXX refers to subdirectory number.

REQUEST PERMDMP,MTHY, ,E.
    Assign dump tape.

UNABLE TO PROCESS REQUEST - ERROR CODE XX

WAITING FOR APF TABLE SPACE

19.1.0 Introduction to BNL ECS

Software modifications to SCOPE to allow ECS usage as an allocatable device were made by Kurt Fuchel, Sid Heller and Graham Campbell of Brookhaven National Laboratory (BNL). Work performed at Brookhaven National Laboratory is supported by the United States Atomic Energy Commission.

A review of the design considerations of the Interim system is contained in BNL Internal Document AMD 503-R (Oct. 1968) The Use of Extended Core Storage in a Multiprogramming Operating System.

For related information see:

Sales Technical Memorandum - ECS Software in SCOPE 3
SCOPE 3 Reference Manual - 60189400
SCOPE 3 Installation Handbook - 60235600
SCOPE Operator's Guide - 60179600
Extended Core Storage Systems Reference Manual - 60225100

ECS is accessed in two ways. Either by job card declaration of an ECS field length; or via stack processor overlay 3SX for I/O requests of ECS files.

In the first case, Direct User Access, the user specifies some field length on his job card, expressed in 1000B units and prefixed by the letters EC. This field length, if available, will be assigned to his control point and he may then use RE and WE instructions to access ECS directly.

Secondly, ECS may be accessed by normal allocatable device I/O, wherein the system handles all of the necessary bookkeeping functions and actual ECS I/O. This is an automatic type access, since the system will attempt to place all allocatable device scratch files in ECS without the user specifying so explicitly. This is called the Allocatable Device Capability.

The Direct User Access capability may be activated without using the BNL ECS mods. To get this type of ECS usage IP.MECS must be set to some non-zero value and selected routines re-assembled in the system. The system will not support multi-computer configurations for this type of ECS usage.

Activating the BNL ECS mods however, will give a user multi-computer support of ECS with both Direct User Access and Allocatable Device Capability. Either or both options may be selected on any or all computers in either single computer or multi-computer configuration.

19.2.0 Direct User Access Capability

Direct User Access capability can be provided very simply and at very little overhead cost to the operating system; the only appreciable time spent by the software is in moving ECS and this is actually less than the time in moving central memory. Since this type of ECS is essentially an extension of central memory, it can be handled in much the same way. Monitor has a storage move and allocation section for ECS. It is actually the storage move code for CM with several flags set to indicate that MTR is really processing ECS. CMR has a storage move loop that reads and writes ECS and is activated in the same manner as the CM storage move program.

Job card declaration of the ECS field length is allowed by modifying 2TJ to recognize and convert ECS field length specifications. ECS is assigned to a control point at job initialization just as CM is assigned by 1RA.

The loader delivers the ECS field length to user programs in X0 just as CM field length is delivered in A0. MEM will allow for the ECS field length to be increased or decreased during job execution. DIS also has the ENFE command, similar to ENFL. ECS, like CM, is returned to the system at job termination by 1EJ.

ECS is divided between the control points on a first-come first-served basis with control point one ECS storage assigned at the low end of ECS, near absolute zero. Since Central Memory can be assigned to control point zero, and this is exactly the length of CMR, so likewise ECS can be assigned to control point zero. This storage below control point one cannot be used by the system for Direct User Access; it can, however be used by other software for this computer or by some other computer, as allocatable device usage for instance. To assign ECS to control point zero requires relatively minor changes to dead-start.

19.3.0 Allocatable Device Usage Capability

Allocatable Device capability is provided by the stack processor overlay 3SX and related routines. This type of usage treats ECS as a zero-latency disk and constructs files in ECS with the same logic as on a normal disk or other allocatable device. That is, Record Block - PRU construction with RBT chains and all the other information normally created for file maintenance is the same for ECS as for other allocatable devices.

When files are first referenced by 4ES, they are examined to see if they qualify for residency on ECS. A list of

file names is kept internal to 4ES containing files which should not be assigned to ECS. The disposition code of the file is checked also and non-zero disposition code files are not assigned to ECS.

Additionally, request cards can be used to assign specific files to ECS. Eventually, if a file is to be assigned to ECS, the FNT is set to indicate this fact by setting the device type to that for ECS. Now, when I/O requests are posted by 4ES in the Request Stack, stack processor will call in the ECS overlay 3SX and perform the I/O.

Since it is not necessary for the program doing the I/O to know which device the file resides on, then for ECS, no changes need be made to such routines as CIO, 2BP, etc.

Some routines need to know about the new device type but only so far as it is included in a general list of device types; they are OPE and REQ.

The only routine that must know exactly how to access ECS is 3SX. It must know the boundaries, size of record blocks, and PRU size, etc.

Record block size is provided by the symbol IP.ECLRB. Boundaries of ECS are set by deadstart with initial values extracted from a section of CMR. So again, deadstart changes are necessary as with Direct User Access capability.

Since Deadstart is where it all begins, let us now look at that portion of the system.

BNL ECS was initially called Interim ECS. It was designed to provide a rudimentary capability for using ECS until such time as a full ECS system became operational. With this in mind and not knowing exactly how ECS should be defined, the original designers planned to provide some flexibility in apportioning ECS between and within computers. This flexibility is provided most easily at deadstart time, by changing the Partition Table.

The Partition Table consists of a set of pointers which divide ECS into the various areas for each computer attached. These areas are then partitioned into blocks for each desired class of usage. The two classes presently supported are Direct User Access and Allocatable Device usage. The most convenient place to keep this table is in the buffer that the ECS code will eventually use. So, at assembly time, the Partition Table is created in T.ICEBUF by calling macros defined in CMR. The mechanics will be discussed later. For now, it is sufficient to say that the RA and FL for each ECS block is stored in a separate word. At deadstart time, a display is brought up showing

these RA and FL fields. They are grouped by computer and labeled appropriately for ease of identification.

Now the aspect of flexibility enters the picture. The operator, by suitable entries, may change any field and adjust for some new partition. There are certain constraints which will be mentioned later. For now it is enough to say that the operator can change any and all fields. This allows various studies to be made on how much ECS is needed for each class of usage and how much ECS is needed for each computer. This facility is provided without the need to reassemble any of the system. ECS may be completely turned off if desired, or if some of ECS should begin to show excessive parities or problems, that part alone may be locked out by changes at deadstart.

The Partition Table in CMR is constructed identically at assembly time, for each computer in a multi-computer installation. When changes are made at deadstart to the Partition Table of one computer the Partition Table of the other computers must also be changed. Re-assembly of the other CMR's is not required, however, since a facility is provided so that each computer, at dead start, can know the most recent Partition Table image in effect.

For example, consider a two computer system. Computer one is deadstarted while the other is idle. The operator makes adjustments to the Partition Table. He then continues deadstart by suitable console commands. The Partition Table image is extracted from the CMR of computer one and written to ECS at absolute zero. Then, as the other computer is deadstarted, a console command causes the Partition Table in its CMR to be ignored and the Partition Table residing in ECS to be read into its CMR overlaying the original Partition Table. The ECS display now shows the most recent Partition Table image. It can be further modified, but care must be exercised so that areas are not overlayed erroneously. Additional computers use the same procedure.

The important feature is that reassembly is not required even though the ECS areas now being used are totally different from those designated at assembly time. This re-reading the Partition Table from ECS may be done across many deadstarts for an indefinite period of time. The standard software will not destroy this area in ECS. Care should be taken that other operating systems particularly customer engineering and experimental systems do not erase the table in ECS. Should the table be over written, however, the problem is not serious. The currently running systems will not reference the Partition Table in ECS, this is done only at deadstart. However, when an attached computer is again deadstarted, it will be necessary to re-

construct the table in ECS. If the required version of the table is different than the assembled version, then the operator makes adjustments to the assembled version by console command. Deadstart is then continued causing this updated Partition Table to be written to ECS. So, the table is written to ECS at each deadstart, unless ECS is turned off. It can be read from ECS at deadstart by suitable command and thus subsequent deadstarts can use the current image which is in ECS.

The flexibility provided by modifications at deadstart has only one drawback. It is not dynamic, and once the system is running, it is impossible to do any further changing of ECS boundaries. This limitation may be removed in some future version of the system.

19.4.0 System Symbols for BNL ECS

BNL ECS modifications are not assembled in the release version of SCOPE. They are assembled by setting the IPARAMS symbol IP.ECNOM non-zero. IP.ECNOM is set to the size of ECS memory that will be used, divided by 100B. This size includes the total ECS for all computers in the configuration. For clarity in all following examples, we shall assume that the ECS memory size is 1,000,000B words, so IP.ECNOM will have the value 10,000B.

Another system symbol relating to BNL ECS usage is contained in IPARAMS. It is IP.ECLRB which is the length of an ECS Record Block in PRU's.

A common deck has been added to the program library. Its name is ECSCOM and it contains most of the system symbol definitions needed for BNL ECS code. The common deck is bounded by a conditional assembly pseudo instruction, the value of IP.ECNOM determines if the code is to be skipped {zero value} or assembled {non-zero value}. A listing of ECSCOM appears below; further discussion of the symbols is deferred until the relevant routines are covered in detail.

ECSCOM

```
COMECS    IFNE    IP.ECNOM,0
*         A COMMON DECK CONTAINING SYMBOLS FOR BNL ECS
*
*         WORDS IN T.ECST, BYTES 0=UNUSED, 1+2=FE, 3+4=RE
*
E.ALLOC   EQU     0                  ALLOCATABLE BLOCK
E.COMM    EQU     1                  COMMON AREA
E.USER    EQU     2                  DIRECT USER ACCESS BLOCK
*
*         THE REST CONTAIN ONLY ONE VALUE, RIGHT ADJUSTED
*
E.STGMVE  EQU     3                  UNUSED
E.LENGTH  EQU     4                  TOTAL LENGTH OF ATTACHED ECS
E.DELAY   EQU     5                  STATISTICS DELAY PERIOD
*
*         WORDS IN T.ECSTAT
*
E.STAT1   EQU     0                  CUMULATIVE SUM OF READ/WRITE TIMES
E.STAT2   EQU     1                  UNUSED
E.STAT3   EQU     2                  NUMBER OF ECS READ OPERATIONS
E.STAT4   EQU     3                  CUMULATIVE SUM OF WORDS READ
E.STAT5   EQU     4                  NUMBER OF ECS WRITE OPERATIONS
E.STAT6   EQU     5                  CUMULATIVE SUM OF WORDS WRITTEN
E.STAT7   EQU     6                  UNUSED
E.STAT8   EQU     7                  UNUSED
*
*         OTHER SYMBOLS
*
E.ECH     EQU     4                  CHANNEL FOR 1SP(3SX)-PP TRANSMISSION
E.PREL    EQU     0                  0=REAL ECS, 1=FAKE ECS(UPPER CM)
E.STDLY   EQU     600D               SYSTEM DEFAULT DELAY PERIOD FOR EC1
N.ECPORT  EQU     1                  NUMBER OF ECS PORTS IN USE
COMECS    ENDIF
```

The remaining system symbols for BNL ECS are contained in SCPTEXT. They are listed below with a brief explanation; they will be explained in detail as the relevant routines are covered.

P.ECST=11B	Word in CMR containing pointers to various ECS tables.
P.LECST=12B	Word in CMR containing lengths of tables specified by P.ECST.

The C. symbols noted below are used to locate bytes in these two pointer word s in CMR.

C.ECFLAW=2	Bytes in CMR which contain address and length of the ECS flaw table.
C.ECST=4	Bytes in CMR which contain address and length of the ECS information table.
C.ECSTAT=1	Bytes in CMR which contain address and length of ECS statistics table.
C.ICEBUF=0	Bytes in CMR which contain address and length of the ICEBOX I/O buffer.
C.LRD=3	Bytes in CMR which contain address and length of the logical record definition table.
E.ERPAR=4001B	Error flag returned by ICEBOX to 1SP-3SX is an ECS parity is encountered.
M.ICE=6	Monitor function, requests monitor to activate ICEBOX.
OV.EC1=3REC1	Symbol defining the name of the ECS statistics accumulation program.

19.5.0 CMR

Basically, CMR has the following construction:

- Pointer Words
- PP Communication Areas
- Control Point Areas
- Central Processor Resident Programs
- Tables
- ECS Tables
- Library

The pointers for the ECS tables and the lengths of the tables are kept in P.ECST and P.LECST. These symbols have the values 11B and 12B respectively. Their format is:

```
P.ECST    EQU   *
          VFD   12/T.ICEBUF/10B
          VFD   12/T.ECSTAT/10B
          VFD   12/T.ECFLAW/10B
          VFD   12/T.LRD/10B
          VFD   12/T.ECST/10B

P.LECST   EQU   *
          VFD   12/L.ICEBUF
          VFD   12/L.ECSTAT
          VFD   12/L.ECLAW
          VFD   12/L.LRD
          VFD   12/L.ECST
```

The T. symbols always have non-zero values because the table origins are always defined. However, the L. symbols actually indicate if the tables have a real length or not. When IP.ECNOM is zero, then the L. symbols have zero value; when IP.ECNOM is non-zero, the L. symbols assume the default values which are defined at the beginning of CMR. They may, however be changed at Installation option.

The Central Processor Resident Programs have the following construction:

Storage Move CM
Storage Move ECS
ICEBOX
Central Exchange Jump Protection

19.5.1 ECS Storage Move

The ECS storage move program performs the moving of the control point ECS memory for Direct User Access in much the same manner as the CM storage move program. It is initiated by MTR when storage needs to be moved. It reads and writes ECS and on the detection of an ECS parity, it sets an error flag in location T.ECSPAR byte 3 and the ECS address/1000B in byte 4. The parity indication is either two or four.

Two indicates that only the program requesting ECS memory is to be aborted. Four indicates that the control point whose ECS memory is being moved is to be aborted in addition to the requesting control point. This occurs if the ECS image gets overwritten during the move and thus there is no guarantee that the moved control point has a valid ECS image any longer.

The error flag for ECS parity on storage move is F.ERECP.

19.5.2 ICEBOX

The ICEBOX section of CMR performs reading and writing of ECS upon request of a PP through the Monitor function M.ICE. Its entry point is at CP.IBX. The PP Message Buffer contains the arguments in the following form:

Word 1	Bytes 1, 2	CM address
	Bytes 3, 4	ECS address/100B
Word 2	Byte 2	Read/Write indicator Read = 0 Write = 1
	Bytes 3, 4	Number of words to transmit

ICEBOX performs the read or write operation, using ECS; or in case E.PREL = 1, a storage move program is activated. The latter occurs if ECS is being simulated in upper CM. This allows installations to configure and test the ECS options even though ECS may not yet be installed.

After the read or write operation has been performed, ICEBOX exits through the EXITLOC code, and accumulates statistics on the ECS operations in the ECS statistics table, T.ECSTAT. These statistics reflect only the allocatable device usage of ECS, not direct user access. The following statistics are accumulated:

Number of reads in T.ECSTAT + E.STAT3
Cumulative sum of words read in T.ECSTAT + E.STAT4
Number of writes in T.ECSTAT + E.STAT5
Cumulative sum of words written in T.ECSTAT + E.STAT6

ECS read parity errors cause entry to RERROR. The ECS read operation will be attempted three times, if parity still persists, the block of ECS will then be read one PRU at a time to find the bad PRU. If no read parity is encountered on the PRU reads, then the error is ignored and an exit to EXITLOC is performed. If the read parity is detected again on the PRU reads, an error flag is set in the second word of the original ICEBOX arguments, byte 0. This error flag is E.ERPAR. The beginning ECS address of the bad PRU is then placed in the flaw table, T.ECFLAW, and the flaw count incremented. The flaw table is first searched to see if this address is already present and if so, it is not duplicated.

The first word of the flaw table contains the flaw count. Flaws are accumulated up to a maximum of L. ECFLAW-1, then on the occurrence of the next flaw, RERROR will set an error flag in word T. ECSPAR, byte 2. When MTR sees this flag set, the comment WARNING..ECS FLAW TABLE FULL is placed in PP zero's message buffer and the system is put into step mode. DSD displays the message and waits for operator acknowledgement and also for the operator to restart the system.

RERROR meanwhile, will place the flaw in the flaw table and reset the flaw count to one. The default length of the flaw table, L. ECFLAW, is 40B on the release tape.

ECS write parity errors are handled as permanent read errors. No attempt is made to re-write ECS. The same error flag is set, E.ERPAR, and the beginning address of the block is placed in the flaw table if it is not already present. It is doubtful if an actual write parity will ever occur, since the hardware does not check for parity errors on write operations. If the error exit is taken on write, then normally this indicates that the bank has been switched to maintenance mode or has lost power.

19.5.3 Construction of the Partition Table.

As mentioned earlier, a Partition Table is constructed in the CMR of each computer which partitions ECS into an area for total system usage and areas for each computer attached to ECS. These areas are then partitioned into blocks for each class of usage. The Partition Table is constructed in T.ICEBUF at assembly time by calling the macros ECBLOCK, ECSPLIT, ECALLOC, and ECUSER which are defined in CMR. (ECSPLIT is used by the other macros and not called by the user directly.) The sizes of ECS blocks are stated in units of 100B words unless specifically noted.

Each computer <u>must</u> have a symbolic label of five characters or less which will be used in the macro calls. This label establishes the relation of the calls to the computer. Labels may be mnemonic, for example, to indicate serial number:

Examples: SER11 SER26 SER28

The blocks into which ECS is partitioned must be in a specific order. Starting at absolute zero and working upward, they must be:

Partition Table	100B	words
Common Block	0	words
Computer 1 allocatable Device Block	m	words
Computer 1 Direct User Access Block	n	words

The last two blocks are repeated for each computer in the system; any m or n may be zero. Each allocatable

19.5.2 ICEBOX

The ICEBOX section of CMR performs reading and writing of ECS upon request of a PP through the Monitor function M.ICE. Its entry point is at CP.IBX. The PP Message Buffer contains the arguments in the following form:

Word 1	Bytes 1, 2	CM address
	Bytes 3, 4	ECS address/100B
Word 2	Byte 2	Read/Write indicator Read = 0 Write = 1
	Bytes 3, 4	Number of words to transmit

ICEBOX performs the read or write operation, using ECS; or in case E.PREL = 1, a storage move program is activated. The latter occurs if ECS is being simulated in upper CM. This allows installations to configure and test the ECS options even though ECS may not yet be installed.

After the read or write operation has been performed, ICEBOX exits through the EXITLOC code, and accumulates statistics on the ECS operations in the ECS statistics table, T.ECSTAT. These statistics reflect only the allocatable device usage of ECS, not direct user access. The following statistics are accumulated:

Number of reads in T.ECSTAT + E.STAT3
Cumulative sum of words read in T.ECSTAT + E.STAT4
Number of writes in T.ECSTAT + E.STAT5
Cumulative sum of words written in T.ECSTAT + E.STAT6

ECS read parity errors cause entry to RERROR. The ECS read operation will be attempted three times, if parity still persists, the block of ECS will then be read one PRU at a time to find the bad PRU. If no read parity is encountered on the PRU reads, then the error is ignored and an exit to EXITLOC is performed. If the read parity is detected again on the PRU reads, an error flag is set in the second word of the original ICEBOX arguments, byte 0. This error flag is E.ERPAR. The beginning ECS address of the bad PRU is then placed in the flaw table, T.ECFLAW, and the flaw count incremented. The flaw table is first searched to see if this address is already present and if so, it is not duplicated.

The first word of the flaw table contains the flaw count. Flaws will be accumulated up to a maximum of 15, then on the occurrence of the 16th flaw, RERROR will cause the system to hang by looping. Since deadstart locks out bad record blocks that exist at dead start time, the occurence of 16 different parity errors indicates ECS is not performing satisfactorily. Call the CE's. The length of the flaw table can be made longer and the code changed in RERROR to allow accumulation of

more than 15 flaws, but this is not the answer to unsatisfactory ECS. The default length of the flaw table, L.ECFLAW, is 40B on the release tape.

ECS write parity errors are handled as permanent read errors. No attempt is made to re-write ECS. The same error flag is set, E.ERPAR, and the beginning address of the block is placed in the flaw table if it is not already present. It is doubtful if an actual write parity will ever occur, since the hardware does not check for parity errors on write operations. If the error exit is taken on write, then normally this indicates that the bank has been switched to maintenance mode.

19.5.3 Construction of the Partition Table.

As mentioned earlier, a Partition Table is constructed in the CMR of each computer which partitions ECS into an area for total system usage and areas for each computer attached to ECS. These areas are then partitioned into blocks for each class of usage. The Partition Table is constructed in T.ICEBUF at assembly time by calling the macros ECBLOCK, ECSPLIT, ECALLOC, and ECUSER which are defined in CMR. (ECSPLIT is used by the other macros and not called by the user directly.) The sizes of ECS blocks are stated in units of 100B words unless specifically noted.

Each computer must have a symbolic label of five characters or less which will be used in the macro calls. This label establishes the relation of the calls to the computer. Labels may be mnemonic, for example, to indicate serial number:

Examples: SER11 SER26 SER28

The blocks into which ECS is partitioned must be in a specific order. Starting at absolute zero and working upward, they must be:

Partition Table	100B	words
Common Block	0	words
Computer 1 allocatable Device Block	m	words
Computer 1 Direct User Access Block	n	words

The last two blocks are repeated for each computer in the system; any m or n may be zero. Each allocatable

device block should normally be an integral multiple of record block size; each direct user access block defined should be an integral multiple of 1000B words and start at an address that is an integral multiple of 1000B. Because the partition table is only 100B words long and because the allocatable device blocks may not be multiples of 1000B words, it is possible that the origins of the direct user access blocks will not fall on the correct boundaries. Two techniques insure that blocks begin at the correct locations.

Preset the Symbol ECRA

The symbol ECRA, internal to the ECBLOCK macro, is incremented by the length of each block as it is defined. The original value of ECRA specifies the first usable word in ECS, and it must be preset before any ECBLOCK calls are made. The minimum value of 1 reserves 100B words of ECS for the partition table. If ECRA is preset to 10B (units are 100B words) and if the allocatable device block defined subsequently is an integral multiple of 1000B words, the proper origin will be set for the first direct user access block defined. This symbol can be incremented or set at any time, therefore block origins may be adjusted.

Example*: ECRA SET 10B ALLOW FOR PARTITION TABLE***

Operator Adjustment at Deadstart Time

At deadstart time, the keyboard entry PACK, will automatically adjust blocks according to specifications and will also close up any gaps between blocks. Only gaps larger than 1000B words are affected. This technique will decrease the field length of some blocks when the original specifications result in non-integral multiples of record sizes for allocatable device blocks or non-integral multiples of 1000B for direct user access blocks. The block size defined for any machine will never be changed.

A common block is defined immediately after the Partition Table. Currently there is no software support for this block; it is defined at this time to allow later support with minimal changes. The length of this block is normally zero, and it is established by calling the ECBLOCK macro.

Example: COMM ECBLOCK 0***

---

* Examples marked with asterisks on the right margin can be used as correction cards to produce a sample configuration system.

Next, the total amount of ECS for each computer is defined by calling the ECBLOCK macro, using the macro call labels and specifying the size of the block. The ECS block includes all ECS used for the computer--both direct user access and allocatable device usage. The ECBLOCK macro does not include the area defined by ECRA.

Example: Assuming the size of ECS is 10000B (units are 100B words), and dividing ECS among three computers using the ECBLOCK macro:

```
SER11  ECBLOCK  4000B  ***
SER26  ECBLOCK  3000B  ***
SER28  ECBLOCK   770B  ***
```

The total for these blocks is 7770B and adding the size of the storage allocated to the partitioning table gives the total size of ECS, 10000B.

After all ECBLOCK macro calls are made, the sub-block sizes are defined; the first is the allocatable device block which, generally, is defined to be an integral multiple of record blocks. Record blocks are generally integral multiples of 100B words. The record block size for ECS is defined by the symbol IP.ECLRB.

This example uses only one record block size for all computers, however this is not a requirement. IP.RECLRB will be set to 30B PRU's. The ECALLOC macro is called as follows:

```
Example:  SER11  ECALLOC  3000B  ***
          SER26  ECALLOC  3000B  ***
          SER28  ECALLOC  0      ***
```

Both SER11 and SER26 now have allocatable device blocks which will hold 100B record blocks. SER28 has a zero-length allocatable device block.

The direct user access blocks are the last sub-blocks defined. These blocks must be an integral multiple of 1000B words and must start at an integral multiple of 1000B. They are defined in two ways:

- The allocatable device block size specified by the ECALLOC macro is subtracted from the total amount of ECS for each computer specified by the ECBLOCK macro.

```
SER11  4000B - 3000B = 1000B
SER26  3000B - 3000B = 0
SER28  770B  - 0     = 770B
```

The ECBLOCK macro defines the symbol LABEL→FL and sets it equal to the total amount of ECS defined for the computer with symbolic name LABEL.

The ECALLOC macro similarly defines the symbol LABEL→1FL and sets it equal to the allocatable device block size.

```
SER11  SER11FL-SER111FL  =  1000B
SER26  SER26FL-SER261FL  =  0
SER28  SER28FL-SER281FL  =  770B
```

When the symbolic form is used, subsequent reconfigurations can be made more quickly, since the direct user access block size need never be specified explicitly it will always be the remainder of the field length after subtracting the allocatable device block size.

Using the first method:

```
Example:  SER11  ECUSER  1000B  ***
          SER26  ECUSER  0      ***
          SER28  ECUSER  770B   ***
```

19.5.4 Other Considerations

In addition to setting up the Partition Table, the EDST and RBR macros must be called for those computers using the allocatable device facility. The EDST macro calls can be as follows:

```
Example for SER11:  AECS  EDST  AX,E.ECH  ***
            SER26:  AECS  EDST  AX,E.ECH  ***
            SER28:  no EDST macro required for ECS
```

Symbols N.DEVICE and N.RBR must be changed to allow definition of ECS as an allocatable device.

```
Example for SER11:  set to previous value plus one
            SER26:  set to previous value plus one
            SER28:  no change required
```

The symbol N.ECPORT must be defined also, as it is the number of ECS ports to be used in a configuration. This symbol can take on values 1-4; it is defined as 1 on the release version of SCOPE.

```
Example:  N.ECPORT  EQU  3  ***
```

A list of identifiers must be defined in CMR starting at T.ICEBUF +15D. These identifiers are used by the operator at deadstart time to verify that the computer is using the correct partition table values. Each identifier is one word, left adjusted display code, maximum of ten characters each. Identifiers should correspond to and occur in the same order as the ECBLOCK calls; they need not differ from the label used for ECS partitioning. The same labels are used in the

following:

```
Example:  ORG    T.ICEBUF  +15D                       ***
          DATA   H$SER11$,H$SER26$,H$SER28$,***
          ,      H$FOR-REAL$
```

Summary

The first usable word in ECS is defined by setting ECRA. The macros ECBLOCK, ECALLOC, and ECUSER are called to construct the Partition table in T.ICEBUF. Macro calls are of the form:

```
LABEL   MACRO-NAME  L
```

Where LABEL is a symbolic label of five characters or less and L is in units of 100B. The communications area must be defined before all other areas.

19.6.0 MTR

ECS is accessed via central processor RE/WE instructions in ICEBOX, a central resident system program. ICEBOX is initiated by MTR when a PP requests the monitor function M.ICE.

The format of the Monitor Request is as follows:

Byte 0 = M.ICE
Byte 4 = 2

B1 in the exchange package is set to the PP Output Register address of the requesting PP. Additional parameters are passed to ICEBOX by the PP in the PP Message Buffer; they are described in Section 19.5.2, ICEBOX.

M.ICE will initiate any Central Executive; byte 4, which is 2 for ICEBOX, specifies which executive to initiate. The M.ICE request is delayed if a system program is already active; MTR initiates only one system program at a time.

When ICEBOX terminates with P =0, MTR calculates the elapsed time and updates two accumulated-time counters.

The first counter is the first word of the ECS statistics table, T.ECSTAT. The second counter is word W.ECTIME in the control point are of the job doing the I/O

19.7.0 Dead Start

As mentioned earlier, the Parition Table changes are allowed only at dead start time. Additionally certain other operations are performed to condition the system for ECS.

19.7.1 STL

STL begins execution in PP1 with the following conditions prevailing:

a. The system has been pre-loaded on an allocatable device.
b. CMR has been loaded into CM.
c. PP's 2-9 are waiting for their respective numbers in CM location zero.
d. PP0 is waiting for a 12B in CM location zero.

When STL exits, MTR and DSD will be executing and PP's 1-8 will be looping in the idle Loop of PP Resident.

STL consists of two parts; the first part is PP Resident and the second part contains the main body of STL dead start code.

STL begins execution at 1000B. After sending PP resident to PP's 1-8 and just before sending MTR and DSD out, STL enters the ECS portion of the program. After initialization, STL enters a loop monitoring central memory for the displays, the keyboard for input, and the central memory QUIT switch for termination.

PP-CP Communication Technique

There are four buffers for communication with IRCP; the Query buffer, the Message buffer, the Parameter buffer, and the Key-In buffer. For the first three of these the first CM word of the buffer is used as a switch. When the low order byte of this word changes, the display is reloaded from central.

The first word of the Key-In buffer is set non-zero by the PP when it writes a keyboard message to IRCP and set to zero by IRCP to indicate the buffer is available.

Operation of the subroutines is as follows:

WTZ Outputs on the display channel from the location specified in the A register until a zero byte is encountered.

DISRP — Displays parameters. Sets initial X and Y coordinates in D.T.1 and D.T.2 then uses DISRP1 to display the parameters for the common area and each ECS port defined in the configuration.

DISRP1 — Outputs coordinates, updates Y-coordinates and uses WTZ to output two words of parameters.

KYBRD — Inputs from the keyboard to a one character per word buffer, displays it and writes it to central memory when CR is entered.

PKKEY — Converts a one character per word buffer into a two character per word buffer.

The format of the display and allowable keyboard entries are described in Section 19.12.0 Deadstart ECS Display.

19.7.2 IRCP

IRCP does many things which are discussed in the Deadstart section of the IMS. Here we shall discuss only the ECS code. After loading, preloading, label checking, etc., and just before terminating, IRCP enters the BNL ECS section of code.

The QUIT switch and pointers to the PP communication buffers have been added at the beginning of the program. These are referenced by STL. We first look for the ECS RBR and its associated EST. If we find them and the EST is invalid, we stop and send the message "ECS EST BAD, CANNOT CONTINUE" through the query buffer to STL. If the RBR is missing the EST is not searched for and processing continues normally. Next we convert the initial allocation parameters to display code and pass them to the PP through the Parameter buffer. We display the query "Tell me about ECS" through the query buffer and loop, monitoring the keyboard buffer for operator entries. When an entry is detected, we clear any message in the message buffer and examine the entry with the TEST macro passing control to the routine for the next possible entry if it is not the current entry.

OFF — The OFF bit is set in the EST, the RBR is set full and its availability reduced to zero and the table T.ECST is zeroed. We then terminate normally.

NONSTD — The values of the Partition Table currently in the first 100B words of ECS are read in

and replace the values currently resident in CM. We then exit to monitor the keyboard.

PACK We enter a loop which sets the RA for the next block to RA+FL of the current block. Then within each block, the allocatable device section is forced to an integral number of record blocks and the remainder put into the user area. Note it is necessary to round allocatable areas to multiples of 100B and user areas to multiples of 1000B.

If the rounded RA of the user area exceeds the RA+FL for the user area, the user area RA will be set to its unrounded value. This could result in a user RA which is not a multiple of 1000B.

This is, in effect, saying that the user area FL is not large enough for at least one block of 1000B words. If this should occur, the user area FL will be set to zero by PACK also. At WRAPUP time the user area RA and FL will be given further consideration.

If FL for a particular machine is zero, the allocatable area FL and the user area FL will be set to zero. The RA for each area will then be set to the RA of the machine, again with the user area RA possibly not being a multiple of 1000B.

In both cases where the user area RA is not a multiple of 1000B, the user area FL will generally be zero. WRAPUP takes this into consideration.

GO If the parameter N.ECPORT is one, the keyboard entry is GO. Otherwise the entry is GO,n. where n is the number of the computer. Before accepting the identification a label obtained from T.ICBUF+15+n is passed through the query buffer and operator verification is requested.

Then the Partition Table is written into the first 100B words of ECS. Now using n (or 1 if N.ECPORT =1) the appropriate entries from the Partition Table are used to fill in the T.ECST table. We now create the RBR for ECS by setting all RB's as locked out, reading the allocatable section and toggling bits in the RBR when an error free RB is read.

The counts of total and error free RB's are used to fill in the availability fields in the RBR header. Then we exit to WRAPUP to turn the system loose.

NUMBER — If the entry is none of the above, then we assume the message is a numeric change to the allocation parameters, but check it carefully. On all errors we assume a keying error and display a message to the operator, through the message buffer, asking him to re-enter the message. We scan the input, converting at most two fields (error otherwise) to binary. If anything other than a numeric, period, or comma is encountered it is an error. Entering more than one word (ten characters) is also an error. The value from the first field is divided by two and used to determine the word to modify, the low-order bit determines if RA or FL is to be modified. After changing the appropriate field, the parameter display is regenerated and changed on the scope. We then exit to the loop monitoring the keyboard.

The WRAPUP procedure, which is entered from GO and OFF, sets the user RA into the control point areas (word W.CPECS) and the exchange packages for all control points.

If the user area FL is zero, the user area RA and FL are both set to zero, thus DSD will not have any FL for the user area for each control point and MTR will also never know that ECS was defined null for the user area. This cleans up the B display of DSD. If either of the user area RA and FL are not multiples of 1000B, they will be rounded down to the next lowest multiple of 1000B. This could result in two or more computers using the same area of ECS, so care must be exercised when using the PACK entry (see discussion on previous pages) and in defining the RA and FL for the user area in CMR. WRAPUP then sets the QUIT switch non-zero and exits from the ECS section of code.

Three subroutines used in this section of coding are:

MTZ — Moves data to zero byte. On entry B1=1, B2=L(FROM), B3=L(TO), B4=Max words; on exit B4 is reduced by the number of words moved. Each byte of the word being moved is examined for zero to terminate the move.

CPAR — Converts the parameters in T.ICEBUF to octal

display code, starting two words into PBUF, for each entry in T.ICEBUF; then increments the PBUF control word and exits.

BTO Discards the low order six bits of X1, converts the next 18 bits to display code (left adjusted with zero fill) in X6 and leaves X1 shifted right 24 bits from its position when this routine was entered.

Footnote to GO Section

The algorithm used for locating bits in the RBR for ECS is as follows:

If:
N = Record block ordinal (starting at zero)
W = Word index (W = 0 is first word of reservation bits)
B = Bit ordinal in word (B = 0 is leftmost bit)

Then using the notation:

$\left(\frac{A}{B}\right)$ is the integral part of the quotient

X_Y is X modulo Y

Let:

$L = N_{100} \quad U = \left(\frac{N}{100}\right)$

Now if $L < 74$ then $W = U$ and $B = L$

Otherwise:

$W = \left(\frac{U}{100}\right) + 40$

$B = (L - 74) + SFTAB\ (U_{10})$

where SFTAB

(0) = 0
(1) = 8
(2) = 12
(3) = 20
(4) = 24
(5) = 32
(6) = 36
(7) = 44

All above numbers except values of SFTAB are in octal.

19.8.0 Stack Processor

As with other allocatable devices, a stack processor overlay has been created for accessing ECS for the Allocatable Device access. The name of this overlay is 3SX. It is structured in a manner similar to the other 3S* overlays with minor differences to account for ECS. Since a PP cannot access ECS directly, a method was created for the information to be transferred to and from ECS. ICEBOX performs this function for 3SX. ICEBOX is a central resident system program, activated by MTR on recognition of the monitor function M.ICE; 3SX need only perform an M.ICE function with appropriate arguments and the data will be transferred between ECS and the central memory buffer, T.ICEBUF.

Unlike other stack processor overlays, 3SX has no head positioning or access time constraints. The sections of code which calculate these variables are null; they return zero-latency and always-positioned status respectively.

When 3SX is initially loaded, it reads CM location P.ECST to obtain the table pointer T.ECST, which is in byte C.ECST. Location T.ECST + E.ALLOC is then read from central memory; it contains the RA and FL for the Allocatable Block of ECS for this computer in the following format:

Byte 0 = unused
Bytes 1 & 2 = FL
Bytes 3 & 4 = RA

The RA for ECS is stored and on subsequent ECS accesses the RA is added to the Logical ECS address to obtain an absolute ECS address for ICEBOX.

The channel normally used by other 3S* drivers for accessing their devices is not needed by 3SX except in the case of PP I/O, when the channel is reserved to enable 1SP to pass data to the requesting PP. The channel is not reserved in any other situation.

The PRU length for ECS is 100B words, the same as other allocatable devices. Since it is convenient to do 18 bit arithmetic in the PP's and since ECS addresses can be up to 21 bits in length, the ECS addresses are divided by 100B to permit ECS addressing in units which coincides with the PRU size. Other allocatable devices usually write more than the 100B words of the PRU. Specifically, two other bytes are included for 6603's and 6638's. One of these bytes holds the PRU

size. This is not possible for ECS because of the 100B addressing scheme. While it is possible to write ECS in other than 100B blocks, it is not elegant to do this in practice. Instead, a Logical Record Definition Table, T.LRD, has been created.

This table contains one bit position for each PRU in ECS. The function of these bits is to specify whether or not the corresponding PRU is 100B words long or a short PRU. When a short PRU is to be written, 3SX stores the word count in the last byte of the 100B word block; sets the corresponding bit in the LRD to one and then calls ICEBOX to write 100B words to ECS.

When a full PRU is to be written, the LRD bit for this PRU is set to 0. When 3SX reads ECS by calling ICEBOX, it reads 100B words, then the LRD is inspected to see if the corresponding bit for this PRU is on or off. If zero {off}, 3SX assumes a PRU length of 100B. If non-zero {on}, 3SX uses the PRU length contained in the last byte of the PRU. The utility routine, XLRDIT, is used to fetch and store the bits in the LRD. The length of the LRD, L.LRD is set at assembly time in CMR: L.LRD EQU IP.ECNOM/100B + IP.ECNOM/1000B. Now IP.ECNOM as defined earlier {Section 19,4.0} is equal to the total size of ECS memory that will be used, divided by 100B. This size includes the total ECS for all computers in the configuration.

This means that for a multi-computer installation, the LRD for each computer will include bits for all other computers; and in fact, only those bits for each individual computer will be used by that computer with a large percentage of wasted space in the LRD.

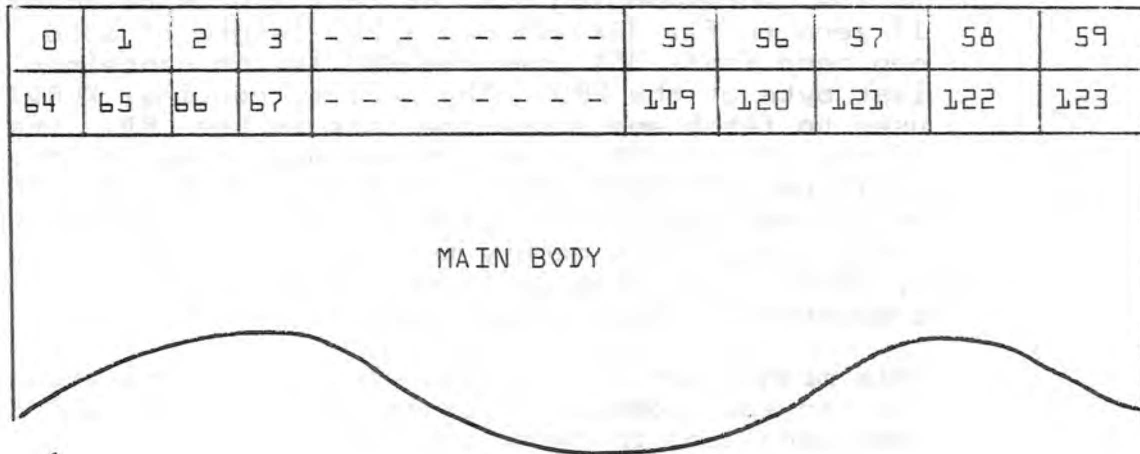
This is necessary evil, however, since as mentioned earlier in dead start, the operator may adjust the various block sizes for each computer, and may specify that some computer will use all of ECS for allocatable device usage. This would require that all the bits in the LRD would be needed by that computer and other computers would not even use the LRD.

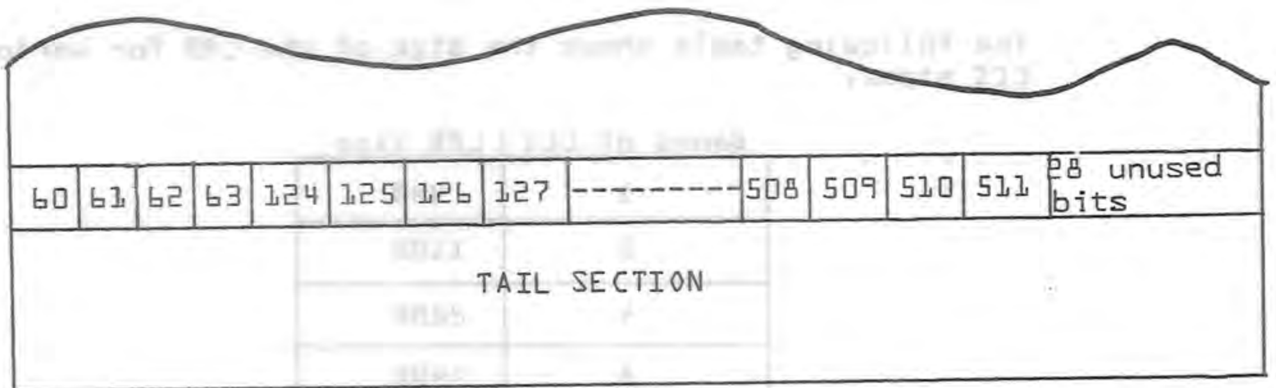
The system is not powerful enough at this time to allow the lengths of tables in CMR to vary at dead start time and to use the extra space, if any becomes available. This shortcoming may be overcome in future versions of the system.

The LRD is constructed for easy indexing. The table is divided into two parts, the main body which contains one CM word for each 64 ECS PRU's and the tail section which contains one CM word for each 512 ECS PRU's.

The 60 bits in word 0 represent logical ECS addresses 0 through 59; the 60 bits in word one represent logical ECS addressed 64 through 123, and so on. Logical ECS address 60 through 63 are represented in word 0 of the tail section. In general, one word of the main body, and four bits of the tail section represent 64 consecutive Logical ECS addresses. For every eight words in the main body, 32 bits in one word of the tail section are used.

Each number in the following table represent a logical ECS address; they are positioned in a 1-1 correspondence with the bits which represent them.





The LRD always begins with logical ECS address 0, regardless of the absolute ECS address.

It is possible to reduce the percentage of unused cells in the LRD by reducing the size of the LRD to only that which is used by each computer. Care must be exercised at dead-start however, that the allocatable device block for any computer is never increased; however a decrease is allowed.

Instead of letting IP.ECNOM be the total ECS memory/100B let it be, for each computer, the allocatable device ECS memory size/100B. Then re-assemble CMR and Stack Processor. Since IP.ECNOM will be non-zero, conditional code will still assemble. The length of the LRD {L.LRD} in CMR will be set to the smaller size and no space will be wasted in the LRD. 3SX will also have the correct size of the LRD so that indexing into the tail section will be correct.

The value of IP.ECNOM must also be used when calling the ECALLOC macro in CMR for the Partition Table definition. For multi-computer configurations the new value of IP.ECNOM may be different for the ECALLOC calls for each computer; therefore the value of IP.ECNOM, and not the symbol IP.ECNOM, is used when calling ECALLOC.

This will somewhat limit the changes that can be made at deadstart time, since increasing the allocatable device block size at dead start will then be an illegal operation and would cause overwriting of the non-existent portion of the LRD during system operation; the Library generally follows this table.

Even if a full LRD is used, the amount of central memory lost is small considering the flexibility and increased performance the system attains with ECS.

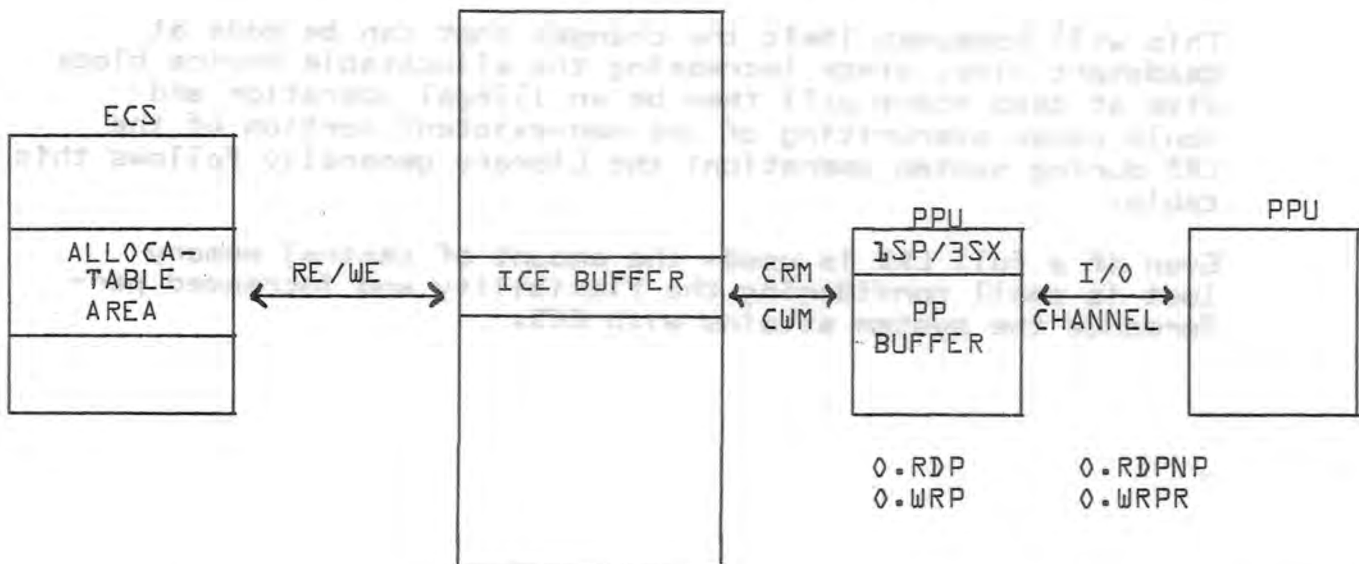
The following table shows the size of the LRD for various ECS sizes.

Banks of ECS	LRD Size
1	44B
2	110B
4	220B
8	440B
16	1100B

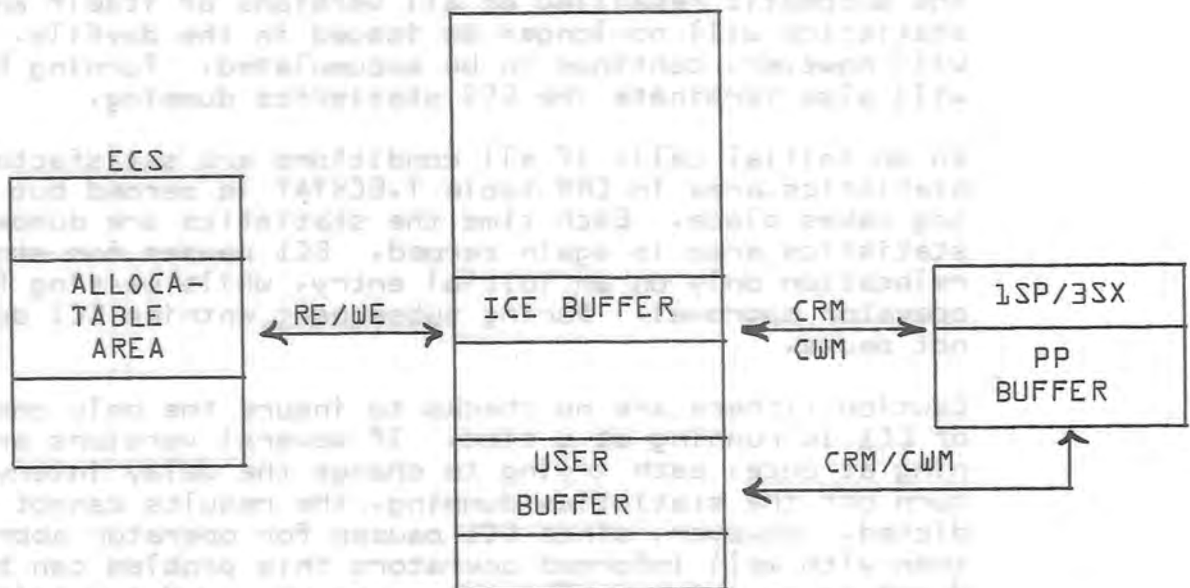
1SP has been organized to do I/O one PRU at a time. For write operations, the PRU is brought into a PP buffer and the 3S* overlay is responsible for sending it to the allocatable device. As mentioned earlier, 3SX does the appropriate LRD and PRU length manipulations and writes the PRU to the central memory buffer, T.ICEBUF. Then the Message Buffer is set up with the appropriate arguments and the monitor function M.ICE requested of monitor. 3SX returns from PP resident after the function has been answered by MTR and ICEBOX has completed the write to ECS.

For read operations, 3SX first requests M.ICE with the appropriate arguments set and returns from PP resident after ICEBOX has read from ECS to T.ICEBUF. Then the PRU is read from T.ICEBUF into the PP buffer and 1SP is responsible for sending the PRU to its final destination.

For PP I/O the PRU is read from, or written to, the requesting PP on the allocatable device channel, hence it is necessary for the data to be in the stack processor PP buffer at some time. The flow for PP I/O is shown below.



Central memory I/O is shown below also.



Notice that for central memory I/O, the data passes needlessly through the stack processor PP buffer. Phase two of BNL ECS will not require this needless passage except in certain special cases.

In any event, after ICEBOX has finished, and 3SX returns from PP resident, error checking is performed. If ICEBOX encountered an ECS parity, it will try to do the ECS operation again, twice. If the error persists, then byte 0 of word two of the Message Buffer will be set non-zero. For ECS parity, its value will be E.ERPAR. 3SX does not attempt the operation again, but assumes a permanent error and calls a 1SP error processing routine to take appropriate action. {See the ICEBOX discussion for related information.} After performing error processing, 3SX returns control to 1SP.

19.9.0 ECS Statistics

This program will dump ECS statistics, which are accumulated by ICEBOX and MTR in the CMR table T.ECSTAT, to the DAYFILE at periodic intervals. For a detailed discussion on what type of statistics and how they are accumulated, see the sections on CMR and MTR. EC1 is called initially by a control card or CPC at some control point. Thereafter, it will be initiated by monitor through the PP delay stack mechanism. The frequency with which it is re-called by monitor is the value of the delay parameter, which is passed initially in the input register. Delay units are one second each.

If delay is zero, E.STDLY will be used. This symbol is defined in ECSCOM. If delay is 7777B, EC1 will terminate the automatic recalling of all versions of itself and ECS statistics will no longer be issued in the dayfile. They will however, continue to be accumulated. Turning ECS off will also terminate the ECS statistics dumping.

On an initial call, if all conditions are satisfactory, the statistics area in CMR table T.ECSTAT is zeroed but no dumping takes place. Each time the statistics are dumped, the statistics area is again zeroed. EC1 pauses for storage relocation only on an initial entry, while waiting for operator approval. During subsequent entries EC1 does not pause.

Caution...there are no checks to insure the only one version of EC1 is running at a time. If several versions are running at once, each trying to change the delay interval or turn off the statistics dumping, the results cannot be predicted. However, since EC1 pauses for operator approval, then with well informed operators this problem can be reduced to a minimum. This approach was used to minimize changes to the operating system, i.e., introducing more flags. One exception to the above is allowed: if one version of EC1 is running, and a terminal call is made, i.e., EC1 {7777}, then all versions will terminate normally.

19.10.0 Other Routines

Miscellaneous other routines have been modified to allow usage of ECS. They include those routines which allow allocatable device and direct user access and are described below very briefly.

19.10.1 REQ

Modifications to REQ allow REQUEST card processing for ECS files. The Hardware mnemonic and device type code for ECS is AX and 20B respectively. A sample REQUEST card for an ECS file is shown below:

REQUEST, file, AX.

19.10.2 4ES

4ES is responsible for assigning scratch files to ECS if space is available. When files are first referenced, 4ES searches the list below for a name match.

PUNCHB	OUTPUT
PUNCH	INPUT
FILMPR	PLOT
FILMPL	HARDPL
	HARDPR

If the file name matches one of the list names, or if the disposition code of the file is non-zero, the file is assigned to some allocatable device other than ECS.

If the name does not match and the disposition code is zero the file is assigned to ECS if space permits. If ECS is full or OFF, the file is assigned to a non-ECS allocatable device.

This scheme is adopted so that files which are generally true scratch files are the only ones assigned to ECS. Files which remain in the system after the job leaves a control point would only tie up ECS until they were disposed of thus decreasing ECS efficiency.

Occasionally scratch files will be assigned to ECS and later made COMMON by the control point. Unfortunately, no protection exists for this type of operation; the COMMON file will remain in ECS. This does, however, allow an EDITLIB, performed after dead start, to effectively use ECS for the COMMON files which contain additions to the system.

19.10.3 <u>1RA, 1EJ</u>

1RA initializes a job at a control point and, if user access ECS is requested on the job card, requests this amount from MTR for assignment to the control point.

1EJ releases user access ECS, which is assigned to a control point, at job termination.

19.10.4 <u>1AJ, Loader</u>

1AJ, PP loader and CP loader pass the user access ECS FL for a control point to a user program by setting X0 to the ECS FL after loading a program, and prior to execution. Similarly, CM FL is passed by setting A0 to the CM FL.

19.10.5 <u>MEM</u>

User calls to MEM will allow increasing, decreasing and checking the size of ECS at a control point. The system macro MEMORY calls MEM through CPC. The SCOPE 3 Reference Manual describes the arguments and action of the MEMORY macro.

19.10.6 <u>DIS, DSD</u>

DIS allows the user access ECS FL to be increased or decreased by the console command ENFE,XXXX.

DSD will display the RA and FL for each control point which has user access ECS. Additionally, the total user access ECS available is displayed on the top of the right screen. See the Operators' Guide for additional specifications.

19.11.0 Related Topics

Some special considerations concerning ECS usage as an allocatable device were not covered in previous sections, since they do not relate to specific routines but rather to the intrinsic qualities of the ECS implementation.

One such consideration is system residence in ECS. Currently only two residences are allowed for any routine in the system, DS for disk and CM for central memory. However, when EDITLIB operations are performed immediately after deadstart or when room is available in the allocatable device block in ECS for scratch files; and routines are added to the system with DS residency, then those routines will be placed in ECS. EDITLIB creates a local file for additions to the system and 4ES will place this local file in ECS if it is not full. When EDITLIB completes the function of adding to the system it leaves this local file in COMMON status.

When system library loading of the routines on the ECS file is performed, the load routine will not know, and need not know, that the file exists in ECS. The load routines will enter stack requests for loading and stack processor alone will be aware that the file is an ECS file and is to be accessed through 3SX. An important feature is that, although the initial residency at deadstart cannot be ECS, it can be changed to ECS residency while the system is running.

The most practical way to accomplish this is to prepare a tape which contains the binary images of routines to be placed in ECS. Then, immediately after deadstart has been completed, delete the selected routines from the system on disk and replace them in the system from the binary tape, by an EDITLIB run. The added routines would be specified to have DS residency but would, in fact be placed in ECS.

In some cases, this process should contribute to improved system performance by eliminating some of the many requests on the system disk for system loading. Not all routines can be placed in ECS, however. Among those excluded are 1SP and all 3S* overlays.

Another system capability relating to ECS usage as an allocatable device is Permanent File Maintenance. ECS implementation allows deadstart changes of the ECS partition to be easily made and it is expected that these changes will occur occasionally. Because permanent files are intended

to be permanent, and changing ECS boundaries would cause files to be lost, permanent file creation in ECS is not allowed. Any attempt to catalog and make permanent a file which resides in ECS will result in the user job being aborted with appropriate error messages.

Another consideration is timing. Allocatable device access of ECS uses the central processor for transferring ECS to and from central memory and although the time actually used by the central processor is small, it does detract from the total amount of central processor time available for user jobs. The amount of this time could be up to ten per-cent of total CP time.

In I/O bound environments this percentage may be higher, but there will be large amounts of CP time available in these instances. CP bound environments, on the other hand, generally do not call for massive data transfers, and the percentage may be smaller.

These considerations are theoretical, however, and each installation will find their own usage of ECS dictated by the primary users of their computers. Emphasis of direct user access ECS for CP bound installations and allocatable device access ECS for I/O bound installations should be a good rule of thumb.

Lastly, there is the problem of memory conflicts. Heavy usage of ECS for direct user access will load central memory correspondingly. Generally, ECS I/O has higher priority for accesses to central memory than PP's. In fact, during ECS I/O, CM is available to the PP's only one cycle per ECS hardware record (8 words). Most PP routines will not be affected adversly by this, since one access per ECS record is permissible under their time constraints. MTR and DSD are among those that are not adversly affected.

Stack Processor, on the other hand, cannot tolerate any delay when accessing central memory. Software and hardware constraints have contributed to very close timing margins and these may be overrun if delays are introduced for central memory accesses. For this reason, during heavy ECS transfers, stack processor may begin to lose revolutions and disk I/O will suffer accordingly.

Stack processor sets the high order bit of the A register for central memory addresses to take advantage of the CMAP option, if it is present. This hardware option provides PP priority for central memory accesses. The value of this option has not been determined at this time and is left as an exercise for the installation.

19.12.0 Deadstart ECS Display

At the top of the left screen on the query line is displayed the message TELL ME ABOUT ECS.

Immediately below is shown a list of various entries the operator may use. As entries are keyed in, they are displayed as the last line on the left screen. Entries are terminated by a period and signaled by pressing the carriage return key. Entries disappear after the carriage return key is pressed whether they are correct or not. An incorrect entry is indicated by an error message appearing one line above the entry line. The error message remains visible until the carriage return key is again pressed at the end of the next entry.

The right screen shows labeling information and values for the partition table. Each entry in the partition table consists of an RA field and an FL field. The entries are displayed, one per line; a pair of numbers to the left of each identifies the fields. Values for each computer designated in the configuration are grouped together and labeled by computer number. A sample display is shown below.

```
                 * ECS PARTITION TABLE *
                    RA          FL

          0,1   000001    000000    COMMON AREA
          2,3   000001    000000    COMMUNICATIONS
          4,5   000001    000000    SPECIAL

          6,7   000001    003777    COMPUTER 1
        10,11   000001    003000    ALLOCATABLE
        12,13   003010    000770    USER

        14,15   004000    004000    COMPUTER 2
        16,17   004000    003000    ALLOCATABLE
        20,21   007000    001000    USER
```

Entries

When the query line is TELL ME ABOUT ECS, the operator may enter any of the following:

OFF. ECS is turned off and is not referenced.

NONSTD. The partition table in the first 100B words of ECS is read and displayed on the right screen.

PACK. The RA's of all areas in ECS are adjusted to make them contiguous. The allocatable device block FL's

are forced to an integral number of record blocks; any remaining storage is put into the succeeding user access block. User access blocks FL's are forced to integral multiples of 1000B beginning at integral multiples of 1000B.

nn,xxxx. Entry nn of the table is changed to xxxx (nn are the numbers displayed on the right screen). All numbers are in octal.

GO. or GO,n. If the IPARAMS symbol N.ECPORT is equal to one, the entry is GO. and the computer is assumed to be one. Otherwise the entry is GO,n. and the computer is assumed to be n.

After the operator enters GO,n. the query line changes to IS THIS *LABEL*. *LABEL*, an identifier preset in CMR, at location T.ICEBUF+15D+n is used for identification and verification of the computer being deadstarted. If the operator responds YES., the normal deadstart procedure continues after internal initialization. If the operator responds NO, the entire process is reinitialized.

Error Messages

If ECS has not been properly defined as an allocatable device or if an ECS parity is detected in the first 100B words of ECS, one of the following messages is displayed and deadstart is discontinued:

ECS EST BAD, CANNOT CONTINUE
ECS PARITY, RESTART AND OFF ECS

Illegal entries will result in one of these messages:

INCORRECT IDENTIFIER, START OVER
ILLEGAL ENTRY

The operator should take action to correct any error displayed.

19.13.0 ECS Partition Table

This table may be changed only when deadstarting by operator action. The partition is determined by a table whose format is:

(each entry)

47	24	23 0
	FL	RA

Word 1   common area {available to all attached computers - to be used for communication}.

Word 2,3   reserved for future development

Word 4   RA, FL for computer 1

Word 5   RA, FL of allocatable section for computer 1

Word 6   RA, FL of user access for computer 1

Words 4, 5, 6 are repeated for each computer attached to ECS.

This table is assembled into the buffer used by ICEBOX and is used only at dead-start {when it is written to ECS}. The first 100B words {absolute} of ECS are reserved for a copy of the latest version of this table.

Several macros are defined to construct this table more easily. Lengths of ECS are in units of 100B words. They are called by:

LABEL ECBLOCK L

where LABEL is a label for a computer to be assigned a block of ECS of Length L. A block of five of these at most {4 ECS ports and 1 common area} must precede the following:

LABEL ECALLOC L

where a length L of ECS is to be devoted to allocatable device usage.

LABEL ECUSER L

A length L of ECS is to be devoted to user access usage.

Starting at T.ICEBUF+15D is a list of identifiers of the computers to be used at dead start time in operator identification of the computer.

19.14.0 Sample Listings

```
*IDENT        SER11EC
*DELETE       ECSCOM.34
N.ECPORT      EQU         3
*INSERT       IPARAMS.17
IP.ECNOM      EQU         10000B
IP.MECS       EQU         40B
*INSERT       CMR.780
AECS          EDST        AX.E.ECH
*INSERT       CMR.852
              BSSZ        38D         ALLOW FOR PRIMARY SYS DEVICE
AECS          RBR         100B
*INSERT       CMR.919
              ORG         T.ICEBUF
*
*             NOW WE DEFINE ECS
*
ECRA          SET         10B
COMM          ECBLOCK     0
*
SER11         ECBLOCK     4000B
SER26         ECBLOCK     3000B
SER28         ECBLOCK     770B
*
*             NOW WE DEFINE SUB-BLOCKS
*
SER11         ECALLOC     3000B
SER26         ECALLOC     3000B
SER28         ECALLOC     0
*
SER11         ECUSER      1000B
SER26         ECUSER      0
SER28         ECUSER      770B
*
              ORG         T.ICEBUF+15D
              DATA        H$SER11$,H$SER26$,H$SER28$,H$FOR-REAL$
*/
*/            DONT FORGET, N.DEVICE AND N.RBR MUST BE INCREMENTED
*/                FOR ECS.
*/            OTHER ROUTINES MUST ALSO BE RE-ASSEMBLED
*/
*COMPILE CMR
```

THE SEQUENCE NUMBERS FOR THE UPDATE CORRECTION CARDS ARE BASED ON A PRE-RELEASE VERSION OF 3.1.6 AND MAY BE DIFFERENT FROM THE RELEASED VERSION OF 3.1.6.

```
*IDENT          SER26EC
*DELETE         ECSCOM.34
N.ECPORT        EQU              3
*INSERT         IPARAMS.17
IP.ECNOM        EQU              10000B
*INSERT         CMR.780
AECS            EDST             AX,E.ECH
*INSERT         CMR.852
                BSSZ             38D      ALLOW FOR PRIMARY SYS DEVICE
AECS            RBR              100B
*INSERT         CMR.919
                ORG              T.ICEBUF
*
*               NOW WE DEFINE ECS
*
ECRA            SET              10B
COMM            ECBLOCK          0
*
SER11           ECBLOCK          4000B
SER26           ECBLOCK          3000B
SER28           ECBLOCK          770B
*
*               NOW WE DEFINE SUB-BLOCKS
*
SER11           ECALLOC          3000B
SER26           ECALLOC          3000B
SER28           ECALLOC          0
*
SER11           ECUSER           1000B
SER26           ECUSER           0
SER28           ECUSER           770B
*
                ORG              T.ICEBUF+15D
                DATA             H$SER11$,H$SER26$,H$SER28$,H$FOR-REAL$
*/
*/              DONT FORGET, N.DEVICE AND N.RBR MUST BE INCREMENTED FOR
*/                 ECS.
*/              OTHER ROUTINES MUST ALSO BE RE-ASSEMBLED
*/
*COMPILE CMR
```

THE SEQUENCE NUMBERS FOR THE UPDATE CORRECTION CARDS ARE BASED ON A PRE-RELEASE VERSION OF 3.1.6 AND MAY BE DIFFERENT FROM THE RELEASED VERSION OF 3.1.6.

```
*IDENT          SER28EC
*DELETE         ECSCOM.34
N.ECPORT        EQU           3
*INSERT         IPARAMS.17
IP.ECNOM        EQU           10000B
IP.MECS         EQU           10B
*INSERT         CMR.852
                BSSZ          38D   ALLOW FOR PRIMARY SYS DEVICE
*INSERT         CMR.919
                ORG           T.ICEBUF
*
*               NOW WE DEFINE ECS
*
ECRA            SET           10B
COMM            ECBLOCK       0
*
SER11           ECBLOCK       4000B
SER26           ECBLOCK       3000B
SER28           ECBLOCK       770B
*
*               NOW WE DEFINE SUB-BLOCKS
*
SER11           ECALLOC       3000B
SER26           ECALLOC       3000B
SER28           ECALLOC       0
*
SER11           ECUSER        1000B
SER26           ECUSER        0
SER28           ECUSER        770B
*
                ORG           T.ICEBUF+15D
                DATA          H$SER11$,H$SER26$,H$SER28$,H$FOR-REAL$
*/
*/              OTHER ROUTINES MUST ALSO BE RE-ASSEMBLED
*/
*COMPILE CMR
```

THE SEQUENCE NUMBERS FOR THE UPDATE CORRECTION CARDS ARE BASED ON A PRE-RELEASE VERSION OF 3.1.6 AND MAY BE DIFFERENT FROM THE RELEASED VERSION OF 3.1.6.

CHAPTER 20 - TABLE OF CONTENTS

20.1 DAYFILE 20-1

20.2 C. E. DIAGNOSTIC ERROR FILE 20-2

20.1 C.E. DIAGNOSTIC PROGRAMS

PL7 Tape

Customer Engineering (C.E.) Diagnostic Programs that execute under SCOPE 3 are contained on the PL7 program library tape which may be obtained from the Distribution Department at Palo Alto. The PL7 tape contains the following programs:

CPU Tests	Memory Tests	Peripheral Equip. Tests
*ALS	*CM6	DT2
*FST	*MY1	MTT
*CT3	EC2	LPT
*CU1		CP1
		CR1
		LP1

These tests are virtually identical with tests of the same mnemonic that are found on the System Maintenance Monitor (SMM) tape. SMM is the system used by customer engineers to run Diagnostic Programs during preventive maintenance periods. IMS material on the above listed tests may be found in the System Maintenance Monitor (SMM) Reference Manual (Pub. No. 60160600). The following test descriptions refer only to internal changes that have been made to certain tests for them to execute under SCOPE 3. See the SCOPE 3 Reference Manual for the external characteristics of each test for running under SCOPE.

A. Random Number Calls to APR

1. ALS, FST, and CT3 call APR for a 12-bit random number each time the test is run.

2. This number is inserted into a byte of the base random number in each test.

   Example: The 12-bit number XXXX is obtained from APR.

   ALS

   Base random number BSN1 is in location 116:

   (116) 1234 5123 4512 3451 2345 B

   XXXX is inserted in the lower byte:

   (116) 1234 5123 4512 3451 XXXX B

   FST

   Base random number CONST is in location 460-463:

```
{460}   0515 0000 1361 1000 0000 B
{461}   6306 0002 1417 1445 0003 B
{462}   0017 0075 0014 7060 0000 B
{463}   3610 3201 4236 4342 0216 B
```

XXXX is inserted into each location:

```
{460}   0515 0000 1361 1000 XXXX B
{461}   6306 0002 1417 XXXX 0003 B
{462}   0017 0075 XXXX 7060 0000 B
{463}   3610 XXXX 4236 4342 0216 B
```

CT3

Base random number RAN is in location 1454:

```
{1454}  7654 5670 1234 3210 4567 B
```

XXXX is inserted in the upper byte:

```
{1454}  XXXX 5670 1234 3210 4567 B
```

3. The random number from APR guarantees that a different sequence of random instructions will be generated each time the sequencer version of the test is run.

B. Pass Counters

Each CPU test is pre-set to run a certain number of passes in SCOPE mode of operation. The pass count limits for each test are:

Test	Tag Name of Limit	No. of Passes
ALS	SEQPASS	4000 B
FST	SEQPASS	5000 B
CT3	SEQPASS	4000 B
CM6	PCX2	40,000 B
MY1	None	10
CU1	None	1
EC2	None	1

The execution time in SCOPE mode for these tests can be varied by varying the number of passes.

C. 6638 Disk File Test - DT2

A programmable READ/WRITE section has been added to this test for use with SCOPE. This section is selected by a parameter in the program call: DT2 {10}

After calling test in this manner, the following occurs:

1. Test writes parameters from PP memory to central memory at $RA+100_8$ as shown below.

2. Test allows operator to change parameters if desired. Enter n.GO to continue.

3. Test reads parameters back from central memory and sets up I/O package accordingly.

4. Test executes.

5. Test checks sense switches.

   a. If sense switch 5 is set, test repeats from step 1.

   b. If sense switch 4 is set, test exits section.

   c. If neither sense switches 4 or 5 are set, test loops on I/O indefinitely.

Parameters

Address	Byte	Descriptions
$RA+100_8$	0	Pattern 0=Don't change 1=Zeros 2=Ones 3=Random
	1	R/W word count 1300_8 Max 502_8 assumed if left blank
	2	Ignore parity errors if non-zero.
	3	No. of words in R/W buffers to transfer to central memory at $RA+200_8$.
	4	Write if zero. Read if non-zero.
$RA+101_8$	0	Not used.
	1	Not used.
	2	Position Increment.
	3	Position Limit.
	4	Position Initial
$RA+102_8$	0	Not used.

Address	Byte	Descriptions
	1	Not used.
	2	Head Group Increment.
	3	Head Group Limit.
	4	Head Group Initial.
RA+103_8	0	Not used.
	1	Not used.
	2	Sector Increment.
	3	Sector Limit.
	4	Sector Initial.

Note 1: The first 30_8 words of Read/Write buffers are written into central memory in all sections of this test during read operations.

Format in Central Memory:

WWWW RRRR XXXX XXXX XXXX

WWWW = Write buffer word.
RRRR = Read buffer word.
XXXX = 0000 on compare.
7777 on miscompare.

Note 2: Sense switch settings control test per Appendix A in SMM Reference Manual.

D. 663X Extended Core Storage Test - EC2

General

EC2 is a test for ECS that was written especially for use with SCOPE 3. This test is not described in the SMM Reference Manual but is similar to Sections 11-16, 18, and 19 of the SMM test called "ECS". EC2 tests ECS by merely writing data patterns to the field length of ECS assigned to the control point and then reading back the patterns. A check is made for aborts and data differences.

Entry and Exit Information

EC2 is called by control card or under DIS by the

mnemonic: EC2. There are no calling parameters. Sense switches are used to indicate how many banks of ECS are present in the system:

1 bank of ECS	-	Switches 1, 2, 3 OFF
2 " "	-	ON Switch 1
4 " "	-	" " 2
8 " "	-	" " 3
16 " "	-	" " 1, 2, 3

If no ECS errors are detected, the program will terminate with no dayfile messages or error output. If ECS errors are detected, the following dayfile messages will appear:

SEQUENCER DIAGNOSTIC/EC2/FAILED.
TYPE X.GO. SEE LINE PRINTER FOR ERROR OUTPUT.

The final message will remain at control point and CP will go into recall until operator types: X.GO. This is to call operator's attention to the error which he might not have noticed if message went straight to dayfile without pausing at the control point.

The error output on the line printer gives:

- the failing ECS absolute address
- the actual data word read
- the data expected
- the logical difference of the two

This format is the same as the SMM ECS test; however, only the first 400 errors are listed.

Other Programs Called

The Automatic Program Sequencer (APR) is called via the RA+1 mechanism. The call is: APR (10,XXXXXX). APR returns the following to relative address XXXXXX:

0000 0000 AAAA BBBB CCCC

where, AAAA - CM RA/100B
BBBB -ECS RA/1000B
CCCC -ECS FL/1000B

Narrative

EC2

The main routine of EC2 is written in FORTRAN Extended. Upon entry, sense switch settings are checked to determine the number of ECS banks in the system. NUMBITS is the total number of bay and bank bits in an ECS address. Then a COMPASS sub-program (ECSTEST) is called to do the actual ECS writes, reads, and compares. ECSTEST will return any error information to the main program which will format the error data for the line printer.

ECSTEST

ECSTEST first calls APR to obtain the ECS RA and FL assigned to the control point. It then tests ECS with eight patterns by doing return jumps to LODBUF to load the appropriate pattern in the output buffer, to WRITEE to write the pattern to ECS, and then to RDANDCK to read data back from ECS and check for errors. The patterns used are: all zeroes, all ones, alternating words of ones and zeroes, alternating words of zeroes and ones, 5-2 pattern, 2-5 pattern, parity pattern (odd-even), parity pattern (even-odd). ECS is written in blocks of 10000B. To change the block size, change the value of WDCOUNT. This value must be divisible by 10B. To change the maximum number of errors that are output on the line printer, change MAXERR. All the arrays in the first 3 DIMENSION statements in EC2 must have the same value as MAXERR.

To use EC2 with the RUN compiler, the mechanism for handling parameters must be changed at locations ECSTEST and STORE.

Subroutines

LODBUFF

Enter with the desired pattern in locations PAT, PAT+1, PAT+2, PAT+3. LODBUFF stores the pattern in OUTBUFF. WDCOUNT must be divisible by 10B for LODBUFF to work.

WRITEE

This subroutine writes OUTBUFF into the entire FL of ECS assigned to the control point. Routine exits to WABORT if a write abort occurs.

RDANDCK

This subroutine reads ECS blocks into INBUFF and compares each word with the expected data found in OUTBUFF. Compare error information is saved. Routine exits to RABORT if a read abort occurs.

WABORT

This subroutine is entered when a write abort is detected. The failing section number, ECS address, and word count is saved and the test is aborted without further checking.

RABORT

This subroutine is entered when a read abort is detected. One word ECS reads are attempted throughout the block under test until the failure is detected. If the read abort cannot be recreated, the intermittent parity error counter, INTPAR, is incremented by one and the routine exits at RAB20 to continue RDANDCK routine at R3A. If the read abort is recreated, the actual data that was read is compared with the expected data at RAB30. If the data does not compare, the abort was caused by a parity error in the data which will be detected when the routine exits to R3A. If the data is the same, then the parity bit itself failed and the failure information is saved at this time because the routine at R3A will not detect this type of failure. In summary, RABORT determines whether the read abort was due to an intermittent parity error that could not be recreated, or a solid parity failure in the data, or a solid failure in the parity bit.

20.2 C. E. Diagnostic Error File

General Description

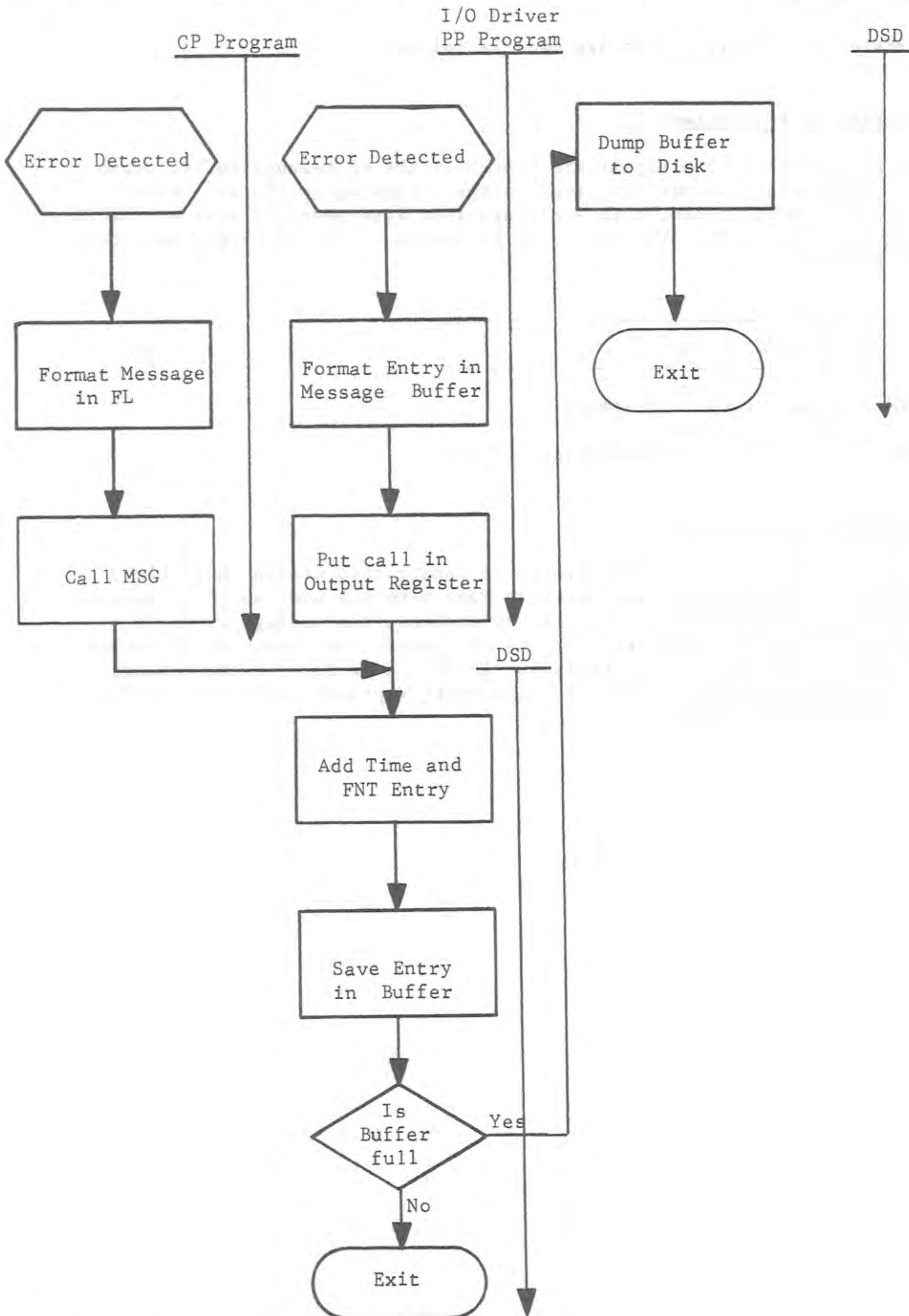
The C. E. Diagnostic Error File is provided as a means of recording system hardware malfunctions detected by the SCOPE Operating System. Errors are recorded via entries from I/O Drivers, PP and CP programs. The entry format, file construction and file maintenance are designed so as to facilitate its utilization in current programs as well as future design efforts.

The file is dynamic from Deadstart Time; thus, its FNT and FET entries are assembled in CMR. If a program detects a hardware malfunction it makes an entry in the file by using one of the formats described in the following sections.

When calls are made via monitor function M.DFM, the entry is placed in the Error File buffer (also in CMR). The dayfile processing routine supplements the entry by adding the time of day, jobname, and FNT information depending on the format used. The dayfile processing routines perform buffer maintenance and dump the contents to disk whenever necessary. An entry count (byte C.HEC of P.HEC in CMR) is maintained, and if the number of entries exceeds a maximum number set by an installation parameter, the operator is notified.

An Analyzer program examines the Error File, collects the statistical data, and outputs the results in usable form for examination by Customer Engineering.

The following diagram shows the technique in which the Error File is implemented:



Entry Calls

Entries to the Error File are made as follows:

Entries by PP Programs

Entries made by PP programs are placed in the PP message buffer using the appropriate format described in the following sections. The PP then sets D.T1 = 0020, D.T2 = message last word address, sets A = M.DFM and does RJM R.MTR. The call will be placed in the PP Output Register as follows:

0001	0020	LWA		

0001 Process Dayfile Message
0020 Flag bit for DSD
LWA LWA of entry in PP message buffer

Entries by CP Programs

Entries made by CP programs are to be constructed within their field length. A call to MSG is placed in RA+1 with the address of the message and a flag (bits 18-23=20) to MSG to indicate the message is to be entered in the Error File. Bits 24-26 specify the length of the entry - the maximum length of the entry is six (6) CM words. If the message is greater than six (6) CM words in length repeated calls must be made by the calling program.

Type 1 - SCOPE SYSTEM I/O DRIVER

<table>
<tr><td>59</td><td></td><td>47</td><td></td><td>29</td><td>23</td><td>11 0</td></tr>
<tr><td colspan="2">a</td><td colspan="2">b</td><td>c</td><td>d</td><td>e</td></tr>
<tr><td>f</td><td>g</td><td>h</td><td colspan="2">i</td><td>j</td><td>k</td></tr>
<tr><td>l</td><td></td><td>m</td><td colspan="2">m</td><td>n</td><td>n</td></tr>
</table>

a	Error Code	Defined under Error Code Descriptions.
b	Program Name	PPU Program making the entry.
c	PPU Number	Computed from contents of D.PPIR
d	Address of Error	Address in program where error was detected
e	FNT address	FNT address of file being used when error was detected
f	EST ordinal	EST ordinal of equipment being driven
g	Channel Number	I/O Channel being used
h	Channel Status	Last 6681 Status returned
i	Equipment Status	Last Equipment Status returned
j	Channel Function Code	6681 Function Code issued
k	Equipment Function Code	Equipment Function Code issued
l	Retry Count	Number of attempts to perform an operation successfully
m	Last Position	For Stack Processor
n	Current Position	For Stack Processor

Type 2 - SCOPE SYSTEM PPU PROGRAMS (Non-I/O)

<table>
<tr><td>59</td><td>47</td><td></td><td>29</td><td>23</td><td>11 0</td></tr>
<tr><td>a</td><td colspan="2">b</td><td>c</td><td>d</td><td>e</td></tr>
<tr><td>f</td><td>f</td><td colspan="2">f</td><td>f</td><td>f</td></tr>
<tr><td>f</td><td>f</td><td colspan="2">f</td><td>f</td><td>f</td></tr>
</table>

a Error Code

b Program Name

c PPU Number

d Address of Error

e FNT Address

f Message - DISPLAY CODED

Type 3 - SCOPE SYSTEM CPU PROGRAM

59	47	35	23	17	11	5 … 0
a	b	b	b		b	(shaded)
(shaded)	(shaded)	(shaded)	(shaded)	c	c	
d	d	d	d		d	
d	d	d	d		d	
d	d	d	d		d	
d	d	d	d		d	

a Error Code

b Program Name 7-Character Program name

c Address of Error

d Message Display Coded message

Type 4 - C. E. DIAGNOSTIC PPU I/O TEST

59		47		29	23	11 … 0
a		b		c	d	e
f	g	h	i		j	k
l	(shaded)	m	m		n	n
o		p	p		p	p

a Error Code
b Program Name
c PPU Number
d Address of Error
e FNT Address
f Logical Unit Number
g Channel Number
h Channel Status
i Equipment Status
j Channel Function Code
k Equipment Function Code
l Retry Count
m Last Position
n Current Position
o Operation Code Last Logical Operation
p Data Error Data

Type 5 - C. E. DIAGNOSTIC Non-I/O PPU TEST

59	47		29	23	11	0
a	b		c	d	e	
f	f	f		f	f	
f	f	f		f	f	

a Error Code
b Program Name
c PPU Number
d Address of Error
e FNT Address
f Message Display Coded

Type 6 - C. E. DIAGNOSTIC CPU PROGRAM

59	47		29		11	0
a	b		c		d	
e	e	e	e	e		
f	f	f	f	f		
f	f	f	f	f		
f	f	f	f	f		
f	f	f	f	f		

a Error Code
b Program Name
c Address of Error
d Pass Count
e Base Random Number
f Display Coded Message

Type 7 - REMOTE TERMINAL (RESPOND, SENTRY, EXPORT/IMPORT, IOD) PROGRAMS

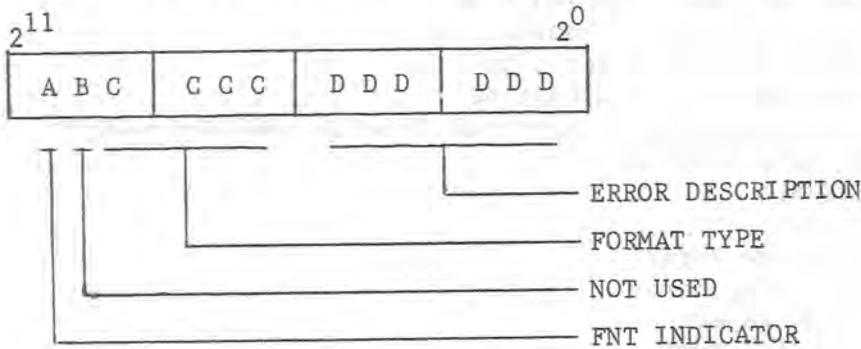
59	47	35	23	11	0
a	b	c	d	e	
f	g	h	i	j	

a	Error Code	
b	Line Number	TTY Line Number
c	LRB	Line Request Byte
d	RBB	Read Buffer Byte
e	WBB	Write Buffer Byte
f	LSB	Line Status Byte
g	TSB	Terminal Status Byte
h	RSB	Reply Status Byte
i	PNB	Port Number Byte
j	PLB	Party Line Byte

Type 8-11 not defined.

Error Codes

The 12-bit error code which is the first byte of every entry is as follows:



A FNT INDICATOR - When set tells the dayfile processing routines that an FNT/FST entry is to be appended to this entry.

B NOT USED.

C FORMAT TYPE - Indicates the type of format used in the entry.

D ERROR DESCRIPTION - General description of the type of error that occurred.

Error Code Definitions

2^{11} Must be set when using format types 1, 2, 4, and 5.

2^{10} Not used.

2^6 - 2^9 Format types as follows:

00	Not used	
01	Type 1	System I/O Driver
02	Type 2	System PPU Program Non-I/O
03	Type 3	System CPU Program
04	Type 4	C.E. Diagnostic PPU I/O Test
05	Type 5	C.E. Diagnostic Non-I/O PPU Test
06	Type 6	C.E. Diagnostic CPU Program
07	Type 7	Control Message (RESPOND or EXPORT/IMPRT)
10	Not used	
11	Not used	
12	Not used	
13	Not used	
14	Not used	
15	Not used	
16	Not used	
17	Not used	

$2^0 - 2^5$ Error Descriptions for Format Type 1

00 Not used.
01 Channel Active - Shouldn't be
02 Channel Inactive - Shouldn't be
03 Channel Full - Shouldn't be
04 Channel Empty - Shouldn't be
05 Non-zero accumulator upon exit from an OAM instruction (Lost Data Suspected)
06 Not used.
07 Not used.
10 Read Parity Error
11 Write Parity Error
12 Function Reject
13 XMSN Parity Error
14 Compare Error
15 Fail to Feed
16 Lost Data
17 RMS Address Error
20 RMS Checkword Error
21 Print Error
22 - 27 Not used.

Error Descriptions for Format types 2-11 not defined.

21.1 DAYFILE PROCESSING

21.2 At deadstart, FNT entries for empty dayfiles are automatically set up at the beginning of the FNT. They are all local and at control point zero; their names are DAYFILE (the system dayfile that records all messages in the system, including those that are sent to the control point dayfiles) and DFILE1, DFILE2, etc., one for each control point except control point 0.

These files do not have normal FET's. Starting at T.DFB in CMR, there is one CM word for each dayfile; at T.DFB+0 for DAYFILE, T.DFB+1 for DFILE1, and so on. Each of these words is a pseudo-FET of the form:

```
VFD     12/A,12/B,12/C,12/D,12/0
```

where C+T.DFB = the address of the first word of the buffer (≠FIRST≠)
B = the length of the buffer (≠LIMIT minus FIRST≠)
A+C+T.DFB= the address of the first unused word of the buffer (A=≠IN minus FIRST≠)
D = the value that A had just before the buffer was most recently dumped. This pointer is not needed for reading and writing the dayfile; but the A-display program of DSD uses it in order to be able to display dayfile messages that have already been dumped to disk.

The dayfile buffers are placed in CMR immediately after these pseudo-FET's.

A PP program issues a dayfile message by putting it in its own message buffer (normally this is done by calling the PP resident sub-routine R.DFM) and an M.DFM request in its output register. A CP program issues a dayfile message by calling PP program MSG, which takes the message from the CP program's field length and then issues it as a PP message.

Monitor will add two CM words at the beginning of every dayfile message:

≠ HH.MM.SS.≠

giving the time in hours, minutes, and seconds; and,

≠NNNNNNN. ≠

where NNNNNNN is the job name taken from the control point area for the calling PP. This additional word can be suppressed by a flag in the M.DFM request; JANUS suppresses it so as not to have ≠JANUS≠ appearing in a message which really relates to a job whose file is being processed by JANUS. In this case, JANUS formats the first word of the message to contain the relevant job name.

When Monitor sees an M.DFM request:

1. If bit 44 of the output register=1, or if the FNT entry for the

corresponding dayfile is in busy status, monitor does nothing. Either of these conditions means monitor has already seen the request and is waiting for the buffer to be emptied before handling the message.

2. Otherwise, monitor determines from B and A (see the explanation of the pseudo-FET above) whether there is room in the buffer for the message. If so, it copies the message into the buffer, advances A by the length of the message, and clears the output register.

3. If there is not enough room in the buffer for the new message, monitor sets the FNT entry for the corresponding dayfile to busy status, and sets bit 44 of the requesting PP's output register to 1. This inhibits further monitor action on the request until bit 44 of the output register is 0 (set by DSD on issuing a stack request to dump the buffer to disk) and the status of the FNT entry becomes idle (set by stack processor when the dumping is complete.)

4. DSD periodically scans the PP output registers looking for dayfile requests in which bit 44 is one. When it finds one, it issues a stack request to write out the buffer, resets bit 44 of the dayfile-requesting PP's output register to 0, sets D=A in the pseudo-FET for the file, and initializes A to 0 in the pseudo-FET. When the buffer has been written to disk, stack processor will set the FNT status to idle, and monitor will then be able to copy the new message into the buffer as in (2) above.

In order to minimize the size of the CMR, the system is arranged so that the dayfile buffers can be of any size (except that the highest-addressed one would have to begin at an address below T.DFB+10000B). Normally, the buffer for DAYFILE is 400B CM words long, as it will be as long as all the other dayfiles put together. The buffers for the first and last control point dayfiles are 40B words long, as the first control point is normally used for JANUS, which does not produce messages for its own control point dayfile, and the last control point is generally left empty except for occasional jobs initiated by operator type-in. The rest of the control point dayfile buffers are 100B words long.

This variability means that whenever a dayfile buffer is dumped, the system cannot simply write out an integral number of PRU's, each one being full of information. Instead, DSD must first fill up the unused words at the end of the buffer with CM words of the form VFD 12/2R+ ,48/0; i.e. each word being a blank line with a '+' format character, which makes it a completely null line if printed. Then DSD makes a stack request to write out the smallest number of PRU's that will at least include the entire buffer. So some extraneous material, besides the null lines, may be included at the end of the last PRU written on each dump.

When a job terminates, 1EJ has to copy the job dayfile onto the end of the job's OUTPUT file, as a single record. If any of the dayfile has already been written to disk, 1EJ rewinds it, reads it, and writes it to the OUTPUT file. Then any dayfile in the dayfile buffer is copied to the OUTPUT file, and the final record is terminated. By consulting the pseudo-FET for the control point, 1EJ finds out how long the dayfile buffer really is, and so it can discard any extraneous matter at the end of each "bufferful". However, the filler words are not filtered out by 1EJ; they do no harm (except to waste a little printer time) as far as printing the dayfile is concerned, and if the OUTPUT file is processed in any other way, the processing program will have to filter them out. Finally, 1EJ sets the dayfile buffer to empty, and the FNT entry to empty.

The system dayfile (DAYFILE) keeps on accumulating until the operator types in "n.DAYFILE,xx." This brings up a PP program (1DF) that:

1. Adds a "dayfile dumped" message to the system dayfile.

2. Issues a special M.DFM request to empty the dayfile buffer by writing its contents to the file on disk.

3. Finds a free slot in the FNT, and inserts in it a copy of the FNT entry for DAYFILE, with its own control point number inserted; then sets the pointers in the original DAYFILE FNT entry at control point zero to show an empty file, on which subsequent dayfile accumulation will take place.

4. If the type-in was "n.DAYFILE,LP." or "n.DAYFILE,CP.", the disposition code of the DAYFILE that is now at control point n is set to 0040B or 0010B, so that the control point can be dropped with the assurance that the file will be printed or punched by JANUS. If the type-in was "n.DAYFILE,MT.", 1DF now requests a tape, and when it has been assigned, gets DAYFILE copied to it by putting the statement "COPYBCD(DAYFILE, TAPE)" into the proper word of the control point area.

None of these treatments of the system dayfile is carried out with reference to the actual length of the system dayfile buffer. So the system dayfile buffer, in contrast to the control point dayfile buffers, must be exactly as long as an integral number of PRU's. Otherwise, extraneous words would get written on DAYFILE, and would not be filtered out by 1DF, or JANUS, or the COPYBCD programs when the dayfile came to be dumped. The system dayfile as dumped will also contain groups of filler words (VFD 12/2R+ ,48/0). These are designed to be absolutely null to a printer, but if the dump is to cards or tape, any program that processes it later has to recognize and ignore these fillers.

When a job terminates, 1EJ has to copy the job dayfile onto the end of the job's OUTPUT file, as a single record. If any of the dayfile has already been written to disk, 1EJ rewinds it, reads it, and writes it to the OUTPUT file. Then any dayfile in the dayfile buffer is copied to the OUTPUT file, and the final record is terminated. By consulting the pseudo-FET for the control point, 1EJ finds out how long the dayfile buffer really is--and so it can discard any extraneous matter at the end of each "bufferful". However, the filler words are not filtered out by 1EJ; they do no harm (except to waste a little printer time) as far as printing the dayfile is concerned, and if the OUTPUT file is processed in any other way, the processing program will have to filter them out. Finally, 1EJ sets the dayfile buffer to empty, and the FNT entry to empty.

The system dayfile (DAYFILE) keeps on accumulating until the operator types in "n.DAYFILE.xx." This brings up a PP program (1DF) that:

1. Adds a "dayfile dumped" message to the system dayfile.

2. Issues a special M.DFM request to empty the dayfile buffer by writing its contents to the file on disk.

3. Finds a free slot in the FNT, and inserts in it a copy of the FNT entry for DAYFILE, with its own control point number inserted; then sets the pointers in the original DAYFILE FNT entry at control point zero to show an empty file, on which subsequent dayfile accumulation will take place.

4. If the type-in was "n.DAYFILE.LP." or "n.DAYFILE.CP.", the disposition code of the DAYFILE that is now at control point n is set to 00400 or 00100, so that the control point can be dropped with the assurance that the file will be printed or punched by JANUS. If the type-in was "n.DAYFILE.MT.", 1DF now requests a tape, and when it has been assigned, gets DAYFILE copied to it by putting the statement "COPYBCD(DAYFILE,TAPE1)" into the proper word of the control point area.

None of these treatments of the system dayfile is carried out with reference to the actual length of the system dayfile buffer. So the system dayfile buffer, in contrast to the control point dayfile buffers, must be exactly as long as an integral number of PRU's. Otherwise, extraneous words would get written on DAYFILE, and would not be filtered out by 1DF, or JANUS, or the COPYBCD programs when the dayfile came to be dumped. The system dayfile as dumped will also contain groups of filler words (VFD 12/2R- -,48/0). These are designed to be absolutely null to a printer, but if the dump is to cards or tape, any program that processes it later has to recognize and ignore these fillers.