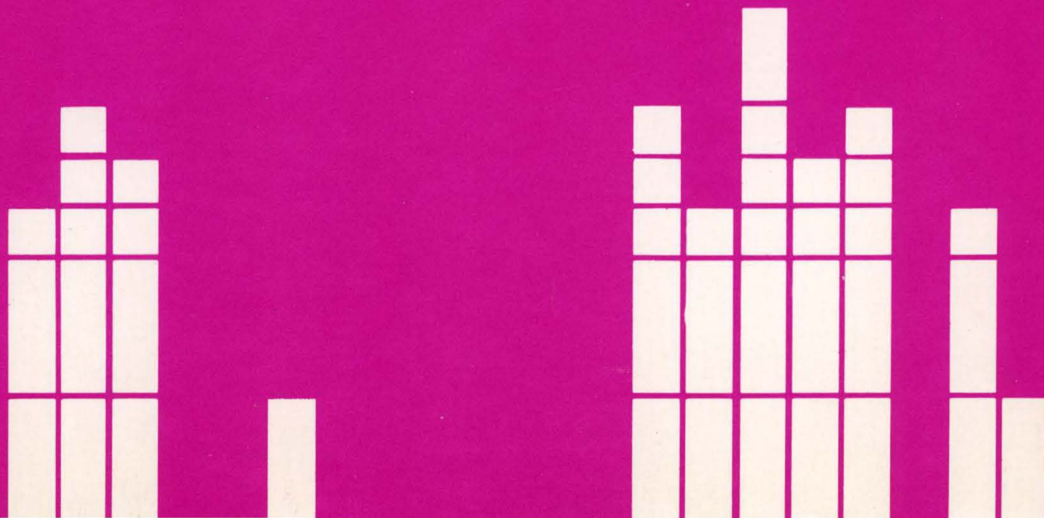
Publication Number
GC31-2066-1

4700 Finance Communication System

Controller Programming Library

Volume 1 General Controller Programming

IBM



4700 Finance Communication System

Controller Programming Library

Volume 1 General Controller Programming

Publication Number
GC31-2066-1

File Number
S370/4300/8100/S34-30

Second Edition (January 1984)

This edition applies to Release 3 and previous releases of the 4700 Finance Communication System and to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters (TNLs).

Changes occur often to the information herein; before using this publication to install or operate IBM equipment, consult the latest *IBM System/370 Bibliography of Industry Systems and Application Programs,* GC20–0370, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. Address comments about this manual to IBM Corporation, Information Development, Department 78C, 1001 W.T. Harris Blvd., Charlotte, NC 28257 USA. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1983, 1984

Preface

This is Volume 1 of the *IBM 4700 Programming Library* - one of a set of six volumes for the 4700 programmer. Figure 0-1 on page v summarizes the topics covered in the other volumes. All six volumes are available from your IBM representative or local branch office under a single order number (GB0F-1387). You will need these volumes if you are developing programs, writing program extensions, or modifying existing programs in the *4700 Assembler Language.*

This book -- *Volume 1: General Controller Programming* -- introduces key programming concepts you need to understand, such as how storage is allocated and how programs can be invoked. This volume also describes the fundamental instructions that form the basis of a typical controller application program. The instructions described in Volume 1 perform basic operations such as adding, subtracting, defining, comparing, and moving data.

Who Should Read This Book

Anyone doing any 4700 Assembler programming will need this book because it contains the instructions that are required in all 4700 programming.

How This Book Is Organized

This book begins with some introductory material about the 4700. It will help you understand how the 4700 operates from a programming point of view. Following the introduction we have grouped instructions according to the overall function they perform, such as data definition and program control, for the purpose of introducing them to the reader. The instructions themselves are arranged alphabetically. We have started each instruction description on a separate page, so that you can reorganize the descriptions in whatever order you later find most useful.

What Else To Read

The first five volumes of *4700 Programming Library* describe the 4700 Assembler Language and explain how you use it. The sixth volume explains how you generate a control program -- CPGEN -- for the controller.

You must generate a control program even if you are not writing your application program in the 4700 Assembler Language. If you are not using the 4700 Assembler Language, you need only the sixth volume. But we recommend you review the other volumes too. They describe many concepts that will enhance your understanding of the sixth volume.

The following is a summary of what you'll find in each volume.

Volume 2 -- Disk and Diskette Programming explains how you organize files for the controller disks and diskettes, and how your program gains access to these files. Both the basic access method and the extended access method are described.

Volume 3 -- Communication Programming shows how your program can communicate with a host computer. Two communication protocols are described: Binary Synchronous Communication (BSC) and Systems Network Architecture (SNA).

Volume 4 -- Loop and Device Cluster Adapter (DCA) Device Programming explains how your program controls and exchanges data with the terminals attached to the controller. Protocols for both loop-attached terminals and for DCA-attached terminals are discussed.

Volume 5 -- Cryptographic Programming describes the cryptographic facilities that are available in the controller and in the 4704 encrypting Personal Identification Number (PIN) keypad.

Volume 6 -- Control Program Generation tells you how to define each controller's resources (application programs, storage, and terminals) and specify how they are to be related. If you do not wish to generate the control program at the host computer, Volume 6 tells you how you can generate the control program at the controller.

The 4700 Assembler Language is the basic programming language of the 4701 Controller. If you are writing programs in a higher-level language (COBOL) or are using IBM-written programs in your controller, see the publications for those products. A brief description of these products and their associated publications appears in the *IBM 4700 System Summary,* GC31-2016.

VOLUME 1: GENERAL CONTROLLER PROGRAMMING (GC31-2066)
- Programming Concepts
- Using the General Programming Instructions
- Coding Rules
- General Programming Instructions (Reference)
- General Machine Instruction Formats
- Parameter List Reference
- Status Codes, Program Check Codes, and Error Messages
- Programming Techniques for 3600 Compatibility

VOLUME 2: DISK AND DISKETTE PROGRAMMING (GC31-2067)
- Basic Disk and Diskette Programming
- Extended Disk and Diskette Programming
- Disk and Diskette Programming Instructions (Reference)
- Disk and Diskette Status Codes
- Disk and Diskette Parameter List Reference
- Disk and Diskette Machine Instruction Formats

VOLUME 3: COMMUNICATION PROGRAMMING (GC31-2068)
- SNA/SDLC Communication Programming
- SNA/SDLC Communication Macros
- SNA/SDLC Communication Instructions (Reference)
- BSC3 Host Communication Programming
- BSC3 Communication Instructions (Reference)
- Communication Status Codes
- Communication Parameter List Reference
- Communication Machine Instruction Formats

VOLUME 4: LOOP AND DCA DEVICE PROGRAMMING (GC31-2069)
- General Protocols for Displays
- 4704 and 3604 Displays
- 3270-Compatible Displays and Devices
- 3606 and 3608 Financial Services Terminals
- General Protocols for Printers
- 4710 and 4720 Printers
- 3610, 3611, and 3612 Printers
- 3615 and 3616 Printers
- 3270-Compatible Printers
- 3624 Consumer Transaction Facilities
- Data Stream Mapping (DATSM) Protocols
- Device Status Codes
- Device Parameter List Reference

VOLUME 5: CRYPTOGRAPHIC PROGRAMMING (GC31-2070)
- Cryptographic Concepts and Facilities
- Enciphering and Deciphering Operations
- Generating and Exchanging Cryptographic Keys
- Authenticating Messages
- Validating and Translating PINs
- Using the Encrypting PIN Keypad
- Host Support Encryption Routines (BDKDPRS and BDKDES)
- Cryptographic Programming Instructions (Reference)
- System Cryptography
- Cryptographic Machine Instruction Formats
- Cryptographic Parameter List Reference
- Cryptographic Program Checks and Status Codes

VOLUME 6: CONTROL PROGRAM GENERATION (GC31-2071)
- Overall View of Control Program Generation (CPGEN)
- Sample CPGEN
- CPGEN Macro Statements (Reference)
- Using the Local Configuration Facility (LCF)
- CPGEN Messages
- LCF Messages
- ALA Configuration Macros

Figure 0-1. 4700 Controller Programming Library (GBOF-1387)

Summary of Amendments

GC31-2066-1 (January, 1984)

This edition reflects the following Release 3 changes:

- The addition of dynamic management of main storage including two new instructions: SEGALLOC and SEGFREE
- Two instructions - Address List (ADRLST) and Indexed Return (IRETURN) that provide new return capability following a branch-and-link operation.

Significant changes and additions to this manual are marked with the same change bars that you see at the left of this summary.

Contents

Chapter 1. Introduction 1-1
The Controller 1-2
The 4700 Terminals 1-2
The Network 1-2
Controller Operation 1-3
Programming the 4700 1-3
  Application Programs 1-4
  Main Storage 1-4
  The Logical Work Station 1-5

Chapter 2. Coding Considerations 2-1
Contents and Purpose 2-1
Application Program Organization 2-1
  Nonrelocatable Programs 2-2
  Relocatable Programs 2-3
  Call Programming 2-3
  Subroutine Programming 2-4
  Overlay Programming 2-4
    Nested Overlay Sections 2-5
    Shared Overlay Sections 2-5
    Nonrelocatable Overlays 2-5
    Relocatable Overlays 2-5
  Using Copy Files 2-5
  Programming Notes 2-6
    Referencing Labels Between Sections 2-6
    Dummy Sections 2-6
Main Storage 2-7
  Managing Storage 2-7
    Managing Storage by System Configuration 2-7
    Managing Storage by Application Programming 2-8
    Storage Management by the Controller 2-11
  Initializing Storage 2-12
  Allocating Two Sets of Registers 2-12
  Shared Storage Control 2-12
Addressing Main Storage 2-13
  Segment-Displacement Addressing 2-13
  Segment Header Addressing 2-15
  Register Addressing 2-16
  Modified Register Addressing 2-17
  Programming Notes About Segment Headers 2-19
    Fixed-Length Fields 2-19
    Variable-Length Fields 2-20
  Programming Notes About Field Delimiters 2-22
    Inserting Delimiters in Fields 2-22
    Processing Messages and Fields 2-22
Logical Work Station Dispatching 2-23
  Dispatching Modes 2-23
  Priority Dispatching 2-23
  Entry Point Priority 2-24
  Gaining Control 2-24
  Releasing Control 2-25
System COPY Files 2-26
Condition and Program Check Codes 2-27
Optional Instructions 2-28
COBOL Considerations 2-29
Use of 3600 Programs 2-29

Chapter 3. 4700 Instruction Categories 3-1
Program Definition Instructions 3-1
  Assembly Definition 3-1
  Section Definition 3-3
Assembly Control Instructions 3-3
  Equates 3-3
  COPY Instruction 3-3
  Controlling Base Registers during Assembly 3-3
Assembly Listing Control Instructions 3-4

Data Definition Instructions 3-4
Defining Constants 3-4
Defining Delimiters 3-4
Defining Dump Parameters 3-4
Defining Masks and Modulus Factors 3-4
Defining Tables 3-4
Defining Fields 3-5
Data Operation Instructions 3-5
Formatting Input Data 3-5
Moving Data within Controller Storage 3-5
Verifying Data 3-6
Data Translation 3-6
Table Lookup 3-7
Packing and Unpacking Data 3-7
Packing Instructions 3-7
Unpacking Instructions 3-7
Compression and Compaction 3-8
Data Compression 3-8
Data Compaction 3-8
String Control Characters 3-9
Data Decompression 3-10
Data Decompaction 3-10
Arithmetic and Logical Instructions 3-10
Arithmetic Operations 3-10
Binary Operations 3-11
Zoned Decimal Operations 3-11
Comparisons 3-12
Logical Operations 3-12
AND and ANDI 3-13
INOR and INORI 3-13
EXOR and EXORI 3-13
Testing Bits 3-13
Setting and Resetting Bits 3-13
Shifting Data 3-13
Program Control Instructions 3-14
Call Programming Instructions 3-14
Passing Data Between Programs 3-14
Instructions that Release Control 3-14
Branch Instructions 3-14
Branch on Condition Code Instructions 3-14
Branch on Bit Switch Instructions 3-15
Branch on Index Instruction 3-16
Branch and Link Instructions 3-16
Instructions that Return Control 3-16
The Execute (LEXEC) Instruction 3-17
Storage Management Instructions 3-17
Other 4700 Instructions 3-17
Storage Initialization Instructions 3-17
Scratch-Pad Instruction 3-17
Timer Control Instructions 3-17
The Dump Instruction 3-18

Chapter 4. Coding Rules 4-1
Syntax Notation 4-1
Interpreting the Syntax Notation 4-1
Specifying Operands 4-2
Labels and Mnemonics 4-6

Chapter 5. 4700 Instruction Descriptions (Alphabetically) 5-1
ADDFLD–Add Field 5-3
ADDFLDL–Add Field Logical 5-5
ADDREG–Add Register 5-7
ADDZ–Add Zoned Decimal 5-9
ADRLST–Return Address List 5-11
AND–AND Field 5-13
ANDI–AND Field Immediate 5-15
APBDUMP–DUMP Segment or File to Diskette 5-17
APCALL–Call Assembler Application Program 5-19
APOPT–Application Program Options 5-23

APRETURN–Return to Calling Program 5-25
BEGIN–Assembly Control 5-27
BRAN–Branch 5-31
BRANL–Branch and Link 5-33
BRANLR–Branch and Link Register 5-35
BRANR–Branch Register 5-37
BRANX–Branch on Index 5-39
CAFLD–Compare Arithmetic Field 5-41
CAFLDL–Compare Arithmetic Field Logical 5-43
CAREG–Compare Arithmetic Register 5-45
CCDI–Compare Character Data Immediate 5-47
CCFLD–Compare Character Field 5-49
CCFXD–Compare Character Fixed 5-51
CCSEG–Compare Character Segment 5-53
COBLCALL–Call a COBOL Application Program 5-55
COMP–Compress and Compact 5-57
COMPTB–Build Compaction Table 5-61
COMPZ–Compare Zoned Decimal 5-65
COPY–Copy Source Code 5-67
CRETN–Conditional Return (COBOL) 5-69
DECOMP–Decompress and Decompact 5-71
DECOMPTB–Build a Decompaction Table 5-75
DEFCON–Define Constant 5-77
DEFDEL–Define Delimiters 5-79
DEFDMP–Define APBDUMP Buffer 5-83
DEFLD–Define Field 5-85
DEFRF–Define a Modified Register Address Field 5-89
DEFSTOR–Define Segment Storage 5-93
DIVFLD–Divide Field 5-95
DIVFLDL–Divide Field Logical 5-97
DIVREG–Divide Register 5-99
DIVZ–Divide Zoned Decimal 5-101
DTACCESS–Data Access 5-103
DTAFREE–Data Free 5-105
EDIT–Edit Monetary Field 5-107
ENDINIT–End Initialization Section 5-111
ENDOVLY–End of Overlay Section 5-113
ENDSEG–End Application Program Section 5-115
EQUATE–Equate a Label to a Value 5-117
ERRLOG–Obtain Statistical Counters 5-119
EXOR–Exclusive OR 5-123
EXORI–Exclusive OR Immediate 5-125
EXPS–Exchange Primary and Secondary Field Pointers 5-127
FCLENTER–Define COBOL Entry Linkage 5-129
FCLEXIT–Define COBOL Exit Linkage 5-131
FINDAP–Find Application Program 5-133
FINISH–End the Application Program 5-137
INITSEG–Initialize Segments 5-139
INOR–Inclusive OR 5-141
INORI–Inclusive or Immediate 5-143
INTMR–Interval Timer 5-145
IRETURN–Indexed Conditional Return 5-151
JUMP–Short Branch 5-153
LCHAP–Change Priority 5-155
LCHECK–Check Status of Station-to-Station Write 5-157
LCONVERT–Convert Binary/Character 5-159
LDDI–Load Data Immediate 5-161
LDFLD–Load Field 5-163
LDFLDC–Load Field Character 5-165
LDFLDL–Load Field Logical 5-167
LDFP–Load Primary Field Pointer 5-169
LDLN–Load Field Length Indicator 5-171
LDRA–Load Register Address 5-173
LDREG–Load Register 5-175
LDSECT–DSECT Definition (BEGIN) 5-177
LDSEG–Load Segment 5-179
LDSEGC–Load Segment Character 5-181
LDSEGLN–Load Segment Length 5-183
LDSFP–Load Secondary Field Pointer 5-185
LEJECT–Eject to a New Page 5-187

LEND–DSECT Definition (End) 5-189
LEXEC–Execute 5-191
LEXIT–End of Processing 5-193
LHRT–Load High-Resolution Counter 5-195
LIFOFF–If Off Then Branch 5-197
LIFON–If On Then Branch 5-199
LLOAD–Load an Overlay Section into Main Storage 5-201
LMERGE–Merge Blocks of Records 5-205
Ln–Level Definition for DSECTS 5-213
LPOST–Post Work Station 5-215
LREAD–Read Station-to-Station Message 5-217
LRETURN–Return after a Branch-and-Link 5-219
LSEEK–Seek (Table Lookup) 5-221
LSEEKP–Extended Seek 5-225
LSEEKPL–Extended LSEEK Parameter List 5-229
LSETOFF–Set Off 5-233
LSETON–Set On 5-235
LSORT–Sort a Block of Records 5-237
LSPACE–Space a Line of Output 5-241
LTIME–Time (Fixed Format) 5-243
LTIMET–Time Table 5-247
LTIMEV–Time (Variable Format) 5-249
LTRT–Translate Input Data 5-253
LTRTBEG–Translate Table Begin 5-265
LTRTENT–Translate Table Entry 5-267
LTRTGEN–Translate Table Configuration 5-269
LWAIT–Wait 5-271
LWRITE–WRITE Station-to-Station Message 5-273
MASK–Mask (For EDIT Instruction) 5-275
MOD–Modulus Factor (For MODCHK Instruction) 5-277
MODCHK–Modulus Check 5-279
MPYFLD–Multiply Field 5-281
MPYFLDL–Multiply Field Logical 5-283
MPYREG–Multiply Register 5-285
MPYZ–Multiply Zoned Decimal 5-287
MVCZ–Move and Convert Zoned Decimal 5-289
MVDI–Move Data Immediate 5-291
MVFLD–Move Field 5-293
MVFLDR–Move Field Reverse 5-295
MVFXD–Move Fixed 5-297
MVFXDR–Move Fixed Reverse 5-299
MVSEG–Move Segment 5-301
MVSEGR–Move Segment Reverse 5-303
OVLYSEC–Define Load Address and Entry Point 5-305
PAKFLD–Pack Field 5-307
PAKSEG–Pack Segment 5-309
PAUSE–Suspend Processing 5-311
PLPCMD–Post-List Processor Commands 5-313
PRINTI–Print Macro Expansion 5-315
REBASE–Restore the Base Register for a DSECT 5-317
SAVEBASE–Save the Base Register for a DSECT 5-319
SCALE–Scale Number 5-321
SCRPAD–Scratch Pad 5-327
SECTION–Section Control 5-339
SEGALLOC–Segment Allocate 5-341
SEGCODE–Application Program Section Identifier 5-343
SEGCOPY–Segment Copy 5-345
SEGFREE–Segment Free 5-349
SELECT–Select Segment 0 5-351
SETFLDI–Set Field Immediate 5-353
SETFPL–Set Primary Field Pointer and Field Length Indicator 5-355
SETSFP–Set Secondary Field Pointer 5-361
SHIFTL–Shift-Left Data in a Register 5-365
SHIFTR–Shift-Right Data in a Register 5-367
SINIT–Start Initialization Section 5-369
STATS–Obtain or Reset Extended Statistical Counters 5-371
STFLD–Store Field 5-379
STFLDC–Store Field Character 5-381
STOVLY–Start Overlay 5-383
STSEG–Store Segment 5-385

STSEGC–Store Segment Character 5-387
SUBFLD–Subtract Field 5-389
SUBFLDL–Subtract Field Logical 5-391
SUBREG–Subtract Register 5-393
SUBZ–Subtract Zoned Decimal 5-395
TABLE–Define Table for LSEEK/LSEEKP 5-397
TSTMSK–Test under Mask 5-399
TSTMSKI–Test under Mask Immediate 5-401
UPKFLD–Unpack Field 5-403
UPKSEG–Unpack Segment 5-405
USEBASE–Use a Base Register for a DSECT 5-407
VERIFY–Verify 5-409
VIEW–VIEW APCALL/APRETURN Stack 5-411

Appendix A. Machine Instruction Formats A-1

Appendix B. COPY Files B-1
DEFAPB B-3
  Segment 14 Fields B-3
DEFCPL B-5
DEFDCP B-5
DEFELP B-6
DEFESP B-6
DEFFAP B-7
DEFGMS B-7
  Segment 15 Fields B-10
DEFINT B-13
DEFMER B-14
DEFREG B-14
DEFRGS B-15
DEFSCA B-15
DEFSCP B-15
DEFSEG B-16
DEFSKP B-17
DEFSMS B-18
  Segment 1 Fields B-21
DEFSOR B-26
DEFTRP B-26
DEFTRT B-27
DEFTSX B-27
DEFVUE B-28

Appendix C. Assembler Error Messages C-1

Appendix D. Program Check Codes D-1

Appendix E. Status Codes E-1

Appendix F. Functions Retained for 3600 Compatibility F-1
Split Programs F-1
  APOPT Instruction: SPLIT Operand F-1
  BEGIN Instruction F-1
    APBNM Operand F-1
    INSNAME Operand F-1
  DEFCON Instruction F-1
  LEXEC Instruction F-1
  LLOAD Instruction F-2
  LSEEK Instruction F-2
  LSEEKP Instruction F-2
  LSEEKPL Instruction F-2
    NOINST F-2
    INST F-2
  OVLYSEC Instruction F-3
    inst-org/const-org Operand F-3
  SECTION Instruction F-3
  SEGCODE Instruction F-3
  SEGCOPY Parameter List F-3
  STOVLY Instruction F-4

TABLE Instruction F-4
Segment Indexing F-4
Indexing Affects on Instructions F-9
BRANX Instruction F-9
LDRA Instruction F-9
MVDI Instruction F-9
MVFXD Instruction F-9
SETX–Enable/Disable Segment Indexing F-10
SETXREG–Set Index Register Number F-13
TESTX–Test for Active Indexing F-15

Appendix G. Program Communication with the System Monitor G-1
Application Program Debugging G-1
Programmable Input Facility G-1
Monitor Restrictions under Programmable Input Control G-2

Bibliography X-1

Index X-3

Figures

0-1. 4700 Controller Programming Library (GBOF-1387) v
1-1. The 4700 Finance Communication System 1-1
5-1. Format of Statistical Counters Returned by ERRLOG 5-120
5-2. Physical Device Address Used by ERRLOG 5-121
5-3. Set Field Pointer Instructions Summary 5-359
5-4. Set Field Pointer Instructions Summary 5-364
5-5. Format 1 Request and Information Returned by STATS 5-373
5-6. Format 2 Request and Information Returned by STATS 5-374
E-1. Status Codes E-2
E-2. Status Codes E-4

Chapter 1. Introduction

The IBM 4700 Finance Communication System is a family of telecommunication products designed for financial institutions and their branches. A typical 4700 system is made up of devices and programs that allow processing to be distributed through the network rather than concentrated at the central site. Message processing and terminal control at a finance communication controller enable the central site to provide support for larger and more complex networks, because less central-site processing is required for each terminal. The processing capabilities of the controller also enable branch operations to continue during failures at the central site or in the communication network. Figure 1-1 shows a sample 4700 system.

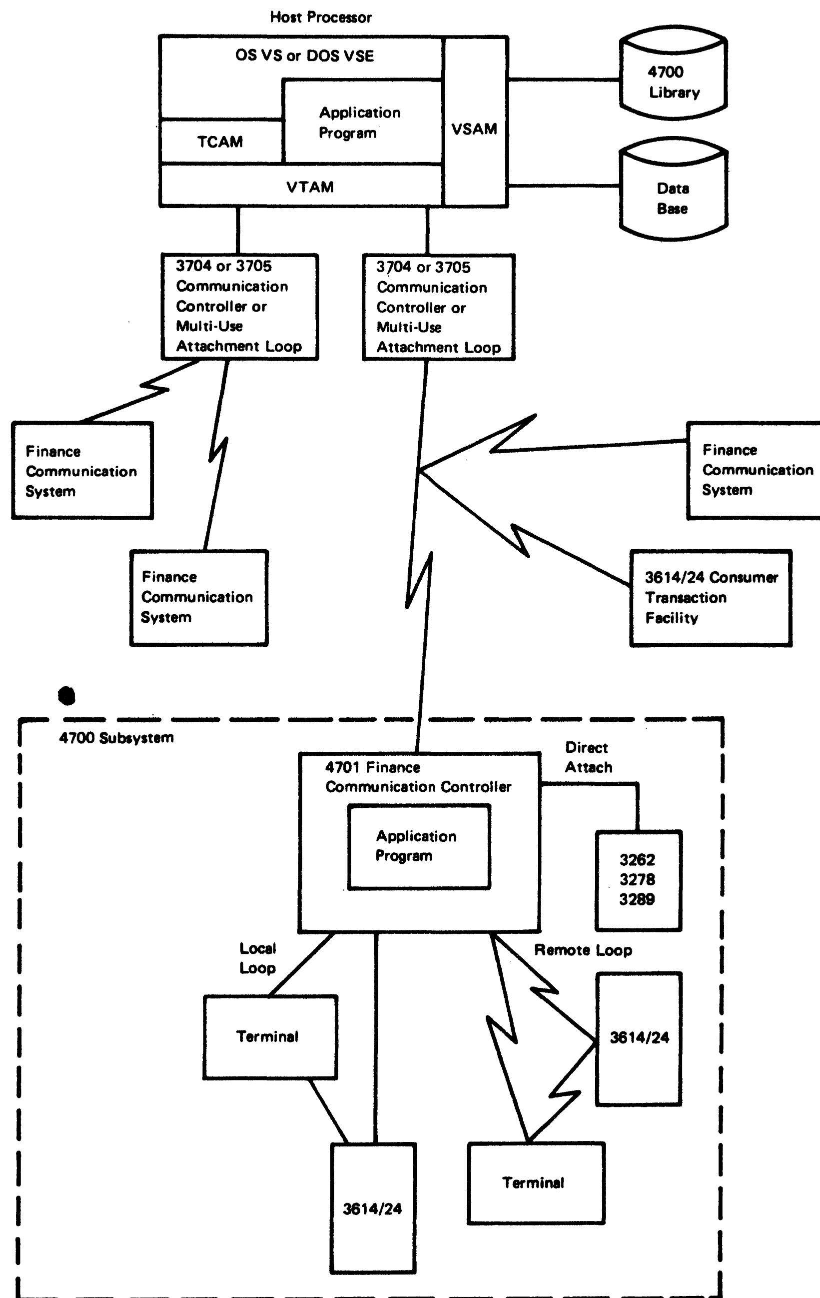


Figure 1-1. The 4700 Finance Communication System

The Controller

The IBM 4701 Controller is a programmable controller that uses application programs, provided by the using institution, to control terminals, process data, and transfer data to the central site. Terminals are attached to the controller either directly, or by a local or remote loop. The controller may be connected to a host processor by several means that are described in *Volume 3: Communication Programming*.

The 4700 Terminals

The terminals that attach to the controller of a 4700 system are a keyboard display, financial services terminals, document and passbook printers, a consumer transaction facility, and several terminals and line printers. Most of the terminals are available in more than one model, providing choices in display size, keyboard size and arrangement, and character sets and printing speeds.

These terminals and a terminal attachment unit that provide telecommunications line connection facilities for 4700 terminals are discussed in Volume 4.

The 4700 terminals are:

IBM 4704 Display
IBM 4710 Receipt/Validation Printer
IBM 4720 Forms/Passbook Printer
IBM 3604 Display
IBM 3606 Financial Services Terminal
IBM 3608 Printing Financial Services Terminal
IBM 3610 Document Printer
IBM 3611 Passbook Printer
IBM 3612 Passbook and Document Printer
IBM 3614 and 3624 Consumer Transaction Facilities
IBM 3615 Administrative Terminal Printer
IBM 3616 Passbook and Document Printer
IBM 3262 Line Printer
IBM 3278 Display
IBM 3279 Display
IBM 3287 Printer

The IBM 3603 Terminal Attachment Unit provides telecommunication line connection facilities for the Financial Services Terminals and other devices. With customer-provided data access arrangements, the 3603 provides manual dial backup capability (over a switched network) to enable you to restore communications if the normal telecommunication line fails. The 3603 is designed for unattended operation and has no impact on programming support.

The Network

The 4700 controller attaches to the host system either directly or on dedicated (leased) or switched lines. You can use: Binary Synchronous Control (BSC3); Systems Network Architecture / Synchronous Data Link Control (SNA/SDLC); CCITT Recommendation X.21; or the 4331 Multiuse Communication Loop protocol. Depending on which protocol is used, the type of link can be either point-to-point or multipoint full-duplex.

You will find additional information on this subject in *Volume 3: Communication Programming*.

Controller Operation

You can operate the controller independently of the central processor. You start the controller by a load operation that, along with many other operations, is carried out at the controller using a facility called the *system monitor*. The system monitor is used by the *control operator*, who, depending on the circumstances, may be one of the branch personnel, an application programmer, or a customer engineer. Information about using the system monitor is in the *IBM 4700 Finance Communication System: Subsystem Operating Procedures.*

You load the controller from a diskette, either the one provided by IBM, referred to as the *installation diskette,* or one you created, referred to as the *operating diskette.* The primary function of the installation diskette is to allow you to create an operating diskette. An operating diskette contains application programs and information to tailor the controller to your needs.

Two configuration procedures are available. One is performed on a host system using the IBM 4700 Host Support (Licensed Program 5668-989) and the other, called the Local Configuration Facility (LCF), runs in the controller.

You must describe your 4700 system using one of the controller configuration procedures. These specifications include information such as: a description of the physical controller, its communication links, and its terminals; and a description of the programming environment to be used.

With the information specified in the configuration procedure as a reference, you can use a 4700 programming language to write an application program. 4700 programming languages consist of symbolic instructions and statements that define data and that become machine instructions.

You can combine configuration data and application programs in a host system and transmit them to the controller to create operating diskettes. You can combine application programs from a disk with configuration data from a diskette to create an operating diskette. You can also transmit controller application programs, without the configuration data, to a controller to replace the current program set on a disk or diskette.

You can also create operating diskettes at the controller by combining LCF-generated configuration data with assembled application programs.

You can copy application programs from disks and diskettes to disks and diskettes.

Programming the 4700

The IBM 4700 Finance Communication System is programmable in ways that are similar to other computing systems. It can be programmed in it's own assembler language, or in COBOL (COmmon Business Oriented Language) a popular programming language for business data processing.

The succeeding sections of this chapter describe, in general terms, 4700 application programs; main storage and how it can be used by application programs; and the concept of a 4700 Logical Work Station.

Application Programs

You may choose to write your own application program or to select from those that are available from IBM. If you choose to write your own application program it may be subdivided and be written in a combination of the two languages mentioned above, that is, part of the program coded in 4700 assembler language and part of it coded in COBOL. You may also separate the program into pieces so that more than one programmer can be assigned or so that sections can be coded at different times. In fact, the organization of the 4700 Programming Library reflects this ability by separating disk and diskette programming from host communication programming and from terminal programming.

In the IBM 4700 Finance Communication System an application program is assigned an amount of main storage and some of the controller's terminals. This combination of an application program, main storage, and terminals, is called a Logical Work Station. One application program can service more than one logical work station, each work station having it's own terminals and it's own storage.

The logical work station, application programs, and controller main storage will be described in more detail in later sections and chapters.

Main Storage

4700 controller main storage will, in general, be used in the following ways: some of it will be used by system functions; some for tables that define the system configuration, and some will be assigned to logical work stations. The amounts of main storage required for each one of these will depend on factors such as; the functions the system is to perform, the numbers and types of terminals, the sizes of application programs, and the amounts of data to be processed. All of these factors will be described in this, and in other volumes of the 4700 Programming Library.

Our primary interest, in this book, is in storage allocated to a logical work station because it is directly related to writing an application program.

The 4700 allocates main storage from storage pools. Storage pools are collections of areas of storage that can be assigned to logical work stations upon request. There are two different types of storage pools in the 4700: pools that are shared by all logical work stations and pools that are owned by one logical work station or shared by a selected group of logical work stations. Chapters 2, 3, and 5 contain more information about main storage pools.

Main storage can be used as private or as shared storage after it has been assigned to a work station.

- *Private storage* is available only to a single logical work station. This assures that information concerning a transaction being processed by a logical work station will not be affected by transactions being processed by other logical work stations. It also assures that one logical work station will not have access to data it does not need.

- *Shared storage* is available to some or all logical work stations. You might use shared storage to transfer an account deposit from a teller logical work station to an account-posting logical work station, or to provide space for information such as a list of overdrawn checking accounts.

Logical work station storage is divided into units called segments. Each logical work station has one segment that contains registers. Another segment contains groups of data fields that provide communication between the controller and logical work stations. The remaining segments are used for data buffers and work areas.

Storage segments are numbered 0 through 15. Segments numbered 0 through 12 are the private storage segments that can be allocated to each logical work station. Segment 13 can be private or shared; Segment 14 always contains an application program; and Segment 15 can be shared among all logical work stations. There can be only one Segment 15 in a system configuration. All of these concepts of 4700 main storage will be further described later.

The Logical Work Station

The 4700 performs work for a conceptual unit called a logical work station. A logical work station is, as we saw earlier, main storage, one or more terminals, and a controller application program. As many as 60 logical work stations can exist in a controller (depending on installation requirements and available storage). All logical work stations have access to disk and diskette drives, to communication links, and to terminals.

The controller allocates processing time to, or *dispatches,* each logical work station in several ways that will be described in this book. When an application program is running on behalf of a logical work station, it can perform various tasks for the financial institution. For example, a commercial bank can have four logical work stations for savings and demand deposit, two for loans, and two for account inquiry. To keep the transaction data separated, each logical work station is given a separate portion of storage. An application program can be shared by more than one logical work station.

Therefore, one could write three application programs: one for savings and demand deposit, one for loans, and one for account inquiry.

The remainder of this manual provides the information needed to design and code a 4700 assembler application program.

The earlier chapters discuss programming considerations such as: selecting a program structure; calling other programs from your program; and designing your program to be called by other programs. They also describe the actual program functions and instructions by categories such as data operations, arithmetic and logical operations, and program control.

The later chapters describe the coding and syntax rules, and the 4700 general controller instructions in detail. Topics such as program checks, status codes, and error messages are in appendixes at the back of this manual.

Chapter 2. Coding Considerations

Contents and Purpose

This chapter contains information about the major concepts of 4700 programming:

- several different ways in which you can organize your program
- getting storage and using it
- addressing data
- getting control of the controller.

The purpose of this chapter is to introduce these concepts to you and to help you begin to know: how the system is designed; what it can do for you; and what you can do with it.

Application Program Organization

The IBM 4700 Finance Communication System provides several ways to organize application programs allowing you to place functions or tasks in different parts of the program; to assign these parts to several programmers; or to implement the parts at different times.

The ways in which you can organize a 4700 application program are:

- Nonrelocatable Programs - programs that have only one part; that is, they are completely contained in one assembly.
- Relocatable Programs - programs made up of sections that can be assembled separately and are link-edited together.
- Call Programming - writing complete programs that are assembled separately and that invoke, that is 'call' other programs or are 'called' by other programs.
- Subroutine Programming - writing routines to which the main program can branch.
- Overlay Programming - writing program *sections* that can be loaded when required.
- Copy Files - sets of instructions that can be assembled into programs or sections.

A non-relocatable program is one assembly containing an entire program. It is the least difficult to understand and use but it also offers the least flexibility. Any change to the program, during development or later, requires that the entire program be reassembled.

A relocatable application program is coded in sections that will be link-edited together by the Host Support. The sections can either be assembled together or assembled separately.

Call programming allows you to divide work into distinct tasks and to write a separate program for each task. To use this facility you must have a primary application program that **calls** secondary application programs. The primary application program must be associated with a logical work station in the configuration specifications. Programs to be called may be resident in main storage or they may be transient, that is; they are on a disk or diskette.

Subroutine programming is a well known technique for coding program functions that are required more than once. For example, you might write a subroutine to handle all disk and diskette operations.

Overlay programming is generally used when available storage is insufficient for an application. In this situation you can use one area of storage for multiple parts of the program when the functions to be performed are somewhat independent of each other. For example, one overlay could receive data from an automated teller terminal; leave it in an area of storage; and a second overlay could send the data to a host system.

Copy files are useful in reproducing sets of instructions in more than one assembly. For example, when 2 programmers are coding related programs and both programs need access to a common data area that can be defined by one set of data definition instructions. The data definition instructions can be created once and then 'copied' into the appropriate assemblies.

Each controller application program includes:

- A BEGIN instruction, which defines the beginning of the application root and builds an application program header that is used by the linkage editor.
- A FINISH instruction that defines the end of the application root.

The BEGIN instruction is usually followed by equates, constants, and data field instructions. These are grouped together to make them easier to find. Instructions that become data must be placed so that they do not interfere with executable machine instructions.

Each application program *section*, other than the root, must begin and end with a SEGCODE-ENDSEG or an OVLYSEC-ENDOVLY pair.

Nonrelocatable Programs

The simplest form of a 4700 application program is one in which the entire program is contained in one assembly. These are usually called 'non-relocatable' programs because there are no parts to arrange before they are loaded into storage. Nonrelocatable programs are preceded by an APOPT instruction that either omits the RELOC operand or specifies RELOC=N. The following shows how a non-relocatable application program can be organized.

```
APOPT RELOC=N
BEGIN
    Data Definition and Machine Instructions
FINISH
END
```

Relocatable Programs

A relocatable application program is coded in sections that will be linkage edited together by the Host Support. 'Relocatable' means that program sections will be placed in order by the linkage editor and that references to data and instructions between sections will be resolved. The application program module being assembled must begin with an APOPT instruction that specifies RELOC=Y.

Each relocatable application program must include a root section, and may include other sections. You must use the following instructions to define application program sections.

- A BEGIN instruction, to name a root section.
- A FINISH instruction, to define the end of a root section.
- A SEGCODE instruction, to name a section and identify the beginning of a section.
- An ENDSEG instruction, to define the end of a section.
- An OVLYSEC instruction, to name a section and identify it as an overlay section.
- An ENDOVLY instruction, to define the end of an overlay section.

Instructions that appear between BEGIN-FINISH, OVLYSEC-ENDOVLY, and SEGCODE-ENDSEG instruction pairs are recognized by the Host Support. If an assembly is attempted where instructions appear outside an application section, Host Support will not process the assembly.

The following shows how a relocatable program can be organized.

```
APOPT  RELOC=Y
BEGIN
              Instructions
FINISH
END
*
APOPT  RELOC=Y
SEGCODE
              Instructions
ENDSEG
END
```

Call Programming

The term 'call' programming simply means that application programs can invoke other application programs. This allows you to assign tasks to small, independent, but interconnected programs. When running in the controller, each program can have its own allocated segment storage, can share segment storage, or both.

As we saw earlier, programs can be written in either of the 2 available languages (Assembler and COBOL). You'll find the instructions that you can use in later chapters. Assembler programs can invoke other assembler programs using the

APCALL instruction, or they can use the COBLCALL instruction to call COBOL programs. COBOL programs can also invoke Assembler programs.

The first program that operates on behalf of a logical work station is the *primary* application program; any program invoked by that program or by any other program is termed a *called,* or a *secondary* application program. Later we will also use the term 'current' application which means the program that is in control of the logical work station.

Application programs may be 'resident', that is, they are in main storage; or they may be 'transient' which means that they are on a disk or diskette.

For example: you might write four programs: one program to handle overall branch office operations; one program to take care of real-time customer activity; one to send account transaction data to a host computer at your home office; and one to save account transactions on a disk. The overall branch office program might:

determine that a teller has a customer deposit and 'call' the customer activity program;

determine that the host link is currently not available;

'call' the disk program to save the transaction data so that it can be sent to the host at a later time.

Subroutine Programming

You can write subroutines to perform discrete functions rather than repeating sequences of instructions in your program. The 4700 has branch-and-link facilities that allow you to pass control to a subroutine. You can also return control to the point at which your program invoked the subroutine. The instructions that you can use are BRANL (Branch and Link); BRANLR (Branch and Link Register); and LRETURN.

Overlay Programming

You must code a root section of an overlay program that remains in main storage at all times. One of its functions must be to cause the non-resident sections to be brought from disk or diskette into main storage as they are needed.

If the application is written in overlays, the overlay sections themselves may be subdivided into sections. The overlay sections and subsections may be assembled separately, together, or in some combination with a root section.

You will find the instructions you can use to write overlay programs among the detailed descriptions later in this book. Specifically, they are: Define an Overlay Section (OVLYSEC); Start an Overlay (STOVLY); End an Overlay (ENDOVLY); and Load an Overlay Section (LLOAD).

The instructions used for overlay programming are:

- LLOAD, which loads the specified application overlay and is in the calling routine.
- STOVLY, which identifies the load point (origin address) of the specified application overlay.
- OVLYSEC, which defines the beginning of the application overlay and may define the entry point of the overlay. It is the first instruction in the overlay section.
- ENDOVLY, which defines the end of the overlay section.

If the application overlay is assembled as relocatable, the following instructions may also be used:

- SEGCODE, which defines the beginning of a section that may be added to the overlay section to form a complete application overlay.
- ENDSEG, which defines the end of a SEGCODE section.

If the controller application program includes application overlays, the first section in each application overlay must begin with an OVLYSEC instruction and end with an ENDOLVY instruction. The remaining sections in the application overlay are those defined by the SEGCODE/ENDSEG instruction pairs. The link-edit function of the Host Support Program resolves all addresses so that the completed application overlay begins at the address of the overlay load point (the STOVLY instruction) and is addressed consecutively.

Nested Overlay Sections

The controller application program may be organized so that nested application overlays are used; that is, one overlay can load another overlay.

Shared Overlay Sections

Your program must handle application overlay areas so that one logical work station does not load an overlay into an overlay area being used by another logical work station.

Nonrelocatable Overlays

Nonrelocatable application overlays are assembled with the application program root but can be loaded individually.

Relocatable Overlays

Relocatable application overlays are similar to nonrelocatable overlays except that they can be assembled separately.

Using Copy Files

Copy files are purely a program assembly facility. You must define a copy file and add it to your macro library in the host system before it can be used by the assembler. When the assembler encounters a COPY instruction it simply inserts all of the contents of the copy file at the point where it found the COPY instruction.

Programming Notes

Here are several things that you should know about writing application programs that have not been described previously.

Referencing Labels Between Sections

Application program sections can refer to labels outside themselves without being assembled together, by using EXTRN statements. If you need to refer to a label that is not defined in this section, you must identify it as an external label with an EXTRN instruction. Another application program section must define the label and identify it with an ENTRY instruction. If these sections are assembled together ENTRY and EXTRN will have no effect; however, you should use these instructions so that you will have the flexibility of being able to assemble the sections separately. ENTRY and EXTRN instructions may appear anywhere within the appropriate application program sections.

Dummy Sections

4700 Assembler Language provides two ways in which you can code a program section that refers to (or uses) data in an area of storage defined in some other section. One way is by a SECTION DUMMY instruction and the other is by an LDSECT (Dummy Section) function. Although they are similar, the LDSECT function offers more flexibility and is the basis for some of the data addressing functions.

Using the SECTION DUMMY instruction, you can refer to, for example, constants that have been defined in the root section of the program. This may sound like the EXTRN and ENTRY description but it differs in that SECTION DUMMY allows you to refer to an entire data area having many labels, not just a few labels in another section.

By using the DEFRF and/or LDSECT instructions you can refer to data areas defined by other sections and other programs. You can dynamically change the reference to the data while the program is running and not be required to recode or reassemble any of the program. This capability will be described in detail later in this chapter.

Main Storage

Managing Storage

The following sections describe three ways that you can manage storage in the 4700:

- Describe main storage usage in your configuration specifications.
- Allow application programs to manage main storage usage dynamically.
- Use a combination of the above.

In the following sections, we discuss managing storage in terms of "defining" and "allocating" storage. "Defining" storage means that you describe an area of storage that a work station or your program may need. "Allocating" storage means that the 4700 assigns an area of storage to an application program or a logical work station.

The term "segment" simply means an area of main storage that is used for specific purposes. Later in this book, we'll talk more about the number of segments that can be used by each work station. For now, we'll use the number 16, 13 private segments and three that can be shared. The purpose of each segment is shown below.

Segment 0 contains the six-byte registers used by your application program for arithmetic and logical operations. As shown later in this chapter, the program can also use these registers for addressing.

Segment 1 is the SMS (System Machine Segment), an area of storage that the system uses for communication with logical work stations.

Segments 2 - 12 are data storage areas assigned to each work station for holding input data, work areas, and output data.

Segment 13 is a data storage area that can be shared with other primary and secondary application programs.

Segment 14 is the read-only storage area holding the application program. It includes constants, machine instructions, and tables. Segment 14 can be a shared segment in the sense that the application program can service more than one logical work station.

Segment 15 is the Global Machine Segment (GMS) shared by **all** application programs.

Managing Storage by System Configuration

Your installation can choose to define and allocate main storage exclusively within the Control Program Generation specifications. Main storage management that is accomplished this way can be called "static", meaning that storage usage can be changed only by modifying and reprocessing your CPGEN.

The system configuration process automatically defines and allocates:

- sixteen 6-byte registers (the first 96 bytes of Segment 0)
- a System Machine Segment for each logical work station (Segment 1)
- Segment 14 for each application program
- a Global Machine Segment.

System configuration can also determine the size of the storage pools that will be in effect during system operation.

An important point to remember is that your installation can take a rigid approach to storage management by defining storage usage only within system configuration specifications. If your installation decides to do so, then you may not need to be concerned with storage management at all within your program. Remember, however, the 4700 allows:

- calling independent programs
- running more than one program on behalf of a work station.

If *any* programs in your controller do these things, you must have some understanding of storage management.

Managing Storage by Application Programming

The 4700 also provides a dynamic way to allocate storage. This storage management function is provided when you include a TRANPL macro in your configuration specifications.

All storage remaining - after requirements for microcode, system configuration data, and resident application programs are satisfied - is available for allocation. When your application program issues an instruction that requires storage, it is allocated from the available space. Storage is returned to the available space when your application program issues the appropriate instruction. The instructions that allocate space are APCALL (Call an Application Program), DTACCESS (Data Access), and SEGALLOC (Segment Allocate). The instructions that release space are APRETURN (Return to Calling Program), DTAFREE (Data Free), and SEGFREE (Segment Free).

Storage Refreshability: The 4700 provides for both non-refreshable (read/write) and refreshable (read-only) programs. You specify which by the Application Program Options (APOPT) instruction. Both kinds of programs can contain executable or non-executable instructions.

Refreshable means that the program cannot be dynamically modified; that it is not an overlay or does not use overlays; and that the 4700 can reuse the storage it occupies and reload the program. If you identify a program as refreshable, the 4700 will not allow it to be modified.

Non-refreshable means that the program can be dynamically modified and that it will remain in main storage until released.

Refreshable programs must be transient and can only exist in the SYSAP data set. These programs will be loaded by the 4700 when another program issues an APCALL or DTACCESS instruction. If your program accesses a non-refreshable program by means of DTACCESS, the 4700 will load that program into a data storage segment; that is, a segment in the range of 2 through 12. Your program can address the program that is loaded only by the register address returned by the DTACCESS instruction.

Storage Management by the 4700: When a request for storage is made, the 4700 will attempt to allocate the space from a storage pool in a series of steps:

1. Search for an available block of storage that can satisfy the request.

2. Move segments currently in use in an attempt to consolidate the available space.

3. Steal the space currently being used by a refreshable area and use this space to satisfy the current request for storage.

If a refreshable area is stolen by storage management it will be automatically reloaded when it is referenced again. This reloading may cause degradation in application program or system performance. Only areas containing programs that are transient and refreshable can be stolen by the system.

To activate the 4700 to perform steal and refresh operations, you must specify a RFSH=Y operand on one of the TRANPL macros in your system configuration. See the *4700 Programming Library Volume 6: Control Program Generation* for further information.

Storage Pools: The 4700 system allows you to define multiple storage pools to tailor the available storage to your specific needs. There are three types of storage pools: station pools, the general pool, and segment-class pools. There can be one station pool for each station or multiple stations can share a single station pool. There is only one general pool. Up to 15 segment-class storage pools can be defined.

Station Pools: Station pools allow you to reserve storage for a station or group of stations. You might define station pools when an application program needs a guarantee that a specific amount of storage be available. For example, a station pool could be defined for the station which runs the 4700 CNM application. This would allow the CNM application to operate as long as no other program is operating for this station and no other station is using the same station pool.

Station pools have the disadvantage that their storage is reserved for a station, or group of stations, and cannot be used by other stations even if some is available.

You define station pool usage by the TRANPL operand of the STATION macro in your system configuration specifications. This pool will be shared only by those stations that identify it in your configuration specifications.

Station pool storage can be used to satisfy both read/write requests and refreshable requests.

The General Pool: You should assign most available storage to the general pool by keeping the station and segment-class pools as small as possible. Any storage not assigned to station pools, to segment-class pools or to the microcode trace area is automatically assigned to the general pool. You can control the amount of space used for the microcode trace area by specifying the TRACE parameter of the STARTGEN configuration macro.

The general pool is shared by all stations. There is no way of excluding a station from using the general pool. Storage in the general pool can be used to satisfy both read/write requests and refreshable requests.

A request is satisfied from the general pool if either there is no station pool defined for the requesting station or there is insufficient space in the station pool. The storage manager will attempt to steal space used by refreshable segments in the station pool **before** attempting to satisfy the request in the general pool.

Segment-Class Pools: Segment-class pools allow you to reserve storage for a particular program or set of programs, independent of the station on which they operate. You might specify a segment-class pool for a program which cannot wait for storage. For example, you may wish to define a segment-class pool for a program which must respond to a request from the host. If you specify the segment-class pool large enough to satisfy this program's storage requirement, and it is the only program that uses the segment-class pool, it will never have to wait for storage.

A segment-class pool is defined by specifying the ID= parameter on the TRANPL macro in your configuration specifications.

If there is insufficient space in the general pool to satisfy a request for storage, an attempt will be made to allocate it from a segment-class pool. Storage is allocated from a segment-class pool to only one station at a time. This station will maintain ownership of the pool until all storage has been returned to the segment-class pool. A request from any other station for storage from that pool will cause the requesting station to be placed into a wait state until the owning station releases the pool or storage becomes available in some other pool.

Each segment-class pool has an ID in the range 1-15. Segment-class pool 1 is used only to satisfy refreshable requests (APCALL for Segment 14 and DTACCESS). Segment-class pools 2-15 can only be used to satisfy the read/write storage requests for Segments 0 and 2-12. Storage requests by DTACCESS and APCALL for non-refreshable programs cannot use segment-class pools. Storage requests are associated with a particular segment-class pool by specifying the ID= parameter on the DEFSTOR instruction or by specifying the segment-class ID in the SEGALLOC parameter list. Your application program should not specify segment-class ID 1.

Usage of segment-class pools is controlled only by programming protocol. There is no enforcement of this protocol by the 4700 controller. If only one program specifies a particular segment-class ID, it will always have sufficient space to complete its operation independent of storage availability in the other pools.

Segment-class storage pools reserve space for particular programs. Because this may adversely affect performance of other programs, segment-class pools should be used with care. Programs that use segment-class pools should attempt to minimize their storage requirements so that these pools can be kept as small as possible.

Storage Management Operation: For a given controller, storage pools can be defined in any combination. If a particular type of pool is not defined, the next type in the hierarchy is used. The steps that the 4700 takes, see "Storage Management by the 4700" on page 2-9, to allocate storage are performed at each level of the hierarchy. The storage allocation hierarchy is illustrated by the following:

1. If a station pool exists and sufficient space can be found, allocate the space.

2. If space can be found in the general pool, allocate the space.

3. If there is a segment-class pool and no program owns this pool and sufficient storage can be found, allocate the space.

4. If there is no segment-class pool for the request or if another program owns this pool or if sufficient space cannot be found,

   a. and if the request specified WAIT=N, set the condition code and process the next instruction.

   b. if the request specified WAIT=Y, place the station in wait state until sufficient space is returned to any pool that can satisfy the request.

When storage is returned to any pool, the wait list is examined. If any pending request can be satisfied, the station is made dispatchable. However, before this station is dispatched, some other station (including the one which just released the storage) may issue a request for this storage. This may cause the station which was waiting to return to its wait for storage. Thus, any wait for storage may be indefinite. If multiple stations are requesting storage and none is available, then potential for a storage deadlock condition exists. A deadlock condition can be avoided by defining segment-class pools that can satisfy the storage requirements of your applications. Any storage wait can be broken by pressing the keyboard Reset key twice. This will cause the APCALL, DTACCESS or SEGALLOC instruction to set a condition code and store attention status in SMSDST.

Storage Management by the Controller

The controller manages storage space for primary and secondary programs when you are using call programming, but you determine how this will be accomplished. You may choose to use one set of storage segments for both the primary program and its secondary programs. If you do, you probably need some conventions to determine which programs use which segments, and when. You may also choose to allow some or all secondary programs to have their own allocations of storage.

Segments allocated to the primary application program and to secondary application programs each have a *segment space ID*. This ID is always 1 for segments allocated to the primary application for each logical work station. The controller assigns a new segment space ID for segments allocated to each secondary application for each logical work station. The *current* segment space ID is the ID of the segments allocated to the current application program.

Initializing Storage

Storage may be initialized, or set to specific values before program run time by specifying the values either during system CPGEN or within your application program. You can use the Start Initialization (SINIT); Initialize Segment (INITSEG); and End Initialization (ENDINIT) instructions to accomplish segment storage initialization in your program. These instructions are described in detail later in this book.

Allocating Two Sets of Registers

When two sets of registers are allocated for a logical work station, the SELECT instruction can be used to select the registers as needed. You must set a segment select character A or B, designating the correct Segment 0, in Segment 1 before you issue SELECT.

For example, if you allow the operator to enter the keyword ADDR, indicating to the program that the operator wants to use the 4704 as an adding machine, you might require the program to use registers other than those used for normal processing. For this purpose, you must allocate two Segment 0's.

There are situations when a logical work station need not be dedicated to one operator (for example, back office applications, such as proof). The controller permits either keyboard or program selection between identical Segment 0's (the registers and optional storage). For program selection, the program must issue the SELECT instruction. SELECT indicates which Segment 0 (A or B) is associated with the station, and allows the work station to address that segment.

Shared Storage Control

Because Segment 15 is accessible to any logical work station, you may need to ensure that one station has exclusive control of the segment for a period of time. You do this by setting an indicator in the global area that the system can check before allowing the station to use the global information or by ensuring that the logical work station does not release control of the controller while using global storage.

For example, all stations may be given control during controller startup so that the tellers can log on. Before allowing most tellers to log on, however, you may wish to ensure that the time and date have been set by one of the tellers. One station sets an indicator in the global storage area that remains on until the time-and-date dialog is completed. As other stations gain control, their application programs can test the indicator and release control until the indicator is off.

Addressing Main Storage

The 4700 addresses data either as the contents of one of 16 registers, or as a field within a storage segment. Registers can be addressed either by referring to the register number or by referring to the register's Segment 0 field. The four ways to address data in the storage segments are:

Segment-displacement addressing address a field by specifying the segment number, the displacement into the segment, and in some cases, the field length. This type of addressing is usually specified as a DEFCON or DEFLD. Its advantage is ease-of-use for fields of fixed location and length.

Segment header addressing use the addressing information, maintained by the system, in the segment header. Each segment has a header containing displacement and length fields that you can use to address data simply by specifying the segment number. The advantages of Segment Header addressing are: you can easily construct data strings of variable length items and you can readily determine the length of items within a data string.

Register addressing load the segment space ID, segment number, displacement, and length information into a register and use the contents of the register to address data. This type of addressing allows your program to dynamically modify addressing values.

Modified register addressing describe data fields using the DEFRF instruction and load addressing information into a register. The controller will modify the register using addressing values from the DEFRF description. This type of addressing has the same advantages as Register Addressing. Also, its use of DEFRF provides a straight-forward way of describing the data area to be processed.

Segment-displacement and segment header addressing do *not* provide a way to specify segment space ID, therefore, you can use them to address only storage that is allocated to your program. Register and modified register addressing allow you to address storage that is allocated to all application programs associated with the same logical work station.

For example, you may use modified register addressing in a secondary application to address storage allocated to the primary application. The primary application can pass its segment space ID to the secondary program allowing it to access segments that were assigned to the primary program.

Segment-Displacement Addressing

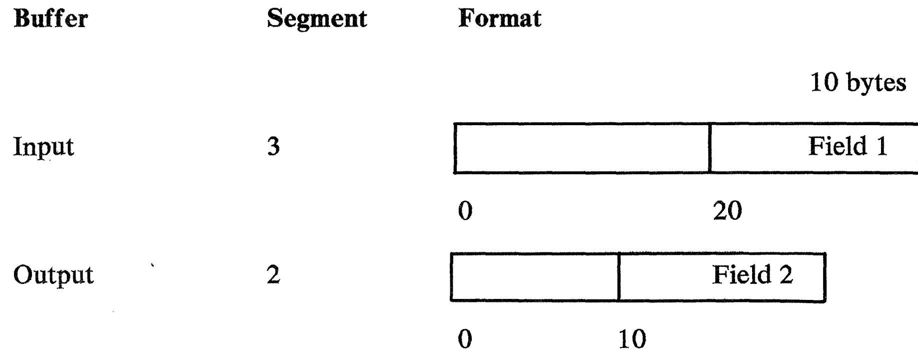
The components of this type of addressing are:

1. the segment number. This is the number (from 0 to 15) of a segment. If the instruction refers to a segment that was never allocated, a program check occurs.

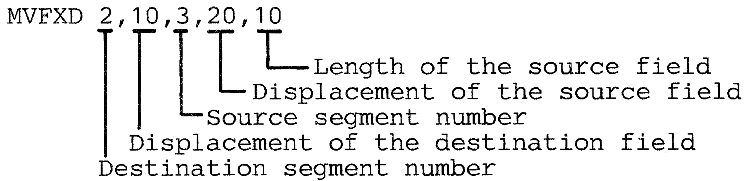
2. the displacement to the beginning of the field, in bytes, from the beginning of the segment. This is a value between 0 and 65 534. If a displacement is specified that is greater than the length of the segment, a program check occurs.

3. the field length, where appropriate. If the displacement plus the length of the field is greater than the size of the segment, a program check occurs.

For example, an input buffer might contain a field that is to be moved to an output buffer for printing.



You can use a MVFXD instruction to move field 1 to field 2:



The following example shows another way of coding the same thing using DEFLD instructions. DEFLD associates a label with a field definition so that subsequent instructions can reference the field symbolically. Application programs written using DEFLDs, rather than absolute references, will be easier to modify.

```
MSG      DEFLD S02,0,10
OUTPT    DEFLD S02,,20
*
DATA1    DEFLD S03,0,20
INPT     DEFLD S03,,10
DATA2    DEFLD S03,,5
         .
         .
         .
         MVFXD OUTPT,INPT
```

The operands of the DEFLD instruction specify the segment, displacement, and length of a field. Notice that the displacement operand has been omitted from several DEFLDs in the example. When you do not specify a displacement, it is equal to the sum of the displacement and length of the last DEFLD referring to that segment. In the example above, the displacement of INPT is 20 and the displacement of DATA2 is 30. You can code either DEFLD or Ln instructions to create DEFLDs (see the LDSECT, Ln, and LEND instruction descriptions).

Segment Header Addressing

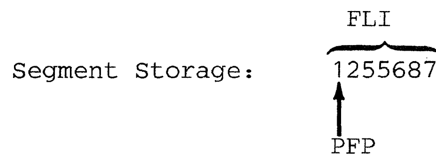
Segment Header addressing uses the segment header associated with each segment. The header consists of:

- The segment length indicator (SLI) contains the length of the segment. This length does *not* include the segment header.
- The primary field pointer (PFP) contains a displacement that is an offset from the beginning of the segment.
- The field length indicator (FLI) contains the length of a field.
- The secondary field pointer (SFP) also contains a displacement that is an offset from the beginning of the segment. Sometimes two pointers, that is, the primary *and* secondary field pointers are required in the same segment.

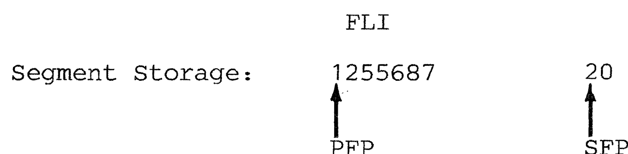
Segment Header addressing applies only to certain instructions, therefore you should be aware of those instructions that allow you to use Segment Header addressing and those that require this type of addressing.

Data entered by a teller or transmitted from the host processor may form fields of variable lengths. Segment Header addressing enables the controller application program to handle these fields.

The PFP and FLI work together to describe a field. For example, a seven-digit account number, located at displacement 0 of the segment, can be addressed when the PFP is equal to 0 and the FLI is equal to 7.

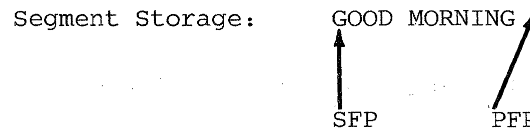


If this account number is to be moved to another location in the same segment, the SFP is used to point to the beginning of the destination field. For example, to move the account number to bytes 20-26, the SFP is set to 20.



You can use a move instruction to move the account number. The number of bytes moved is determined by the value of the FLI.

The PFP and SFP are used to describe a field when a write is performed using the two-byte form of the LWRITE instruction. The SFP points to the leftmost byte of the field, and the PFP points 1 byte past the rightmost byte of the field. For example:



This arrangement is used because move instructions, which are normally used to load the output area, update the PFP. At the end of a move operation, the PFP points 1 byte past the end of the destination field, thus allowing a series of move instructions to place data in a contiguous area without resetting the PFP.

The segment header cannot be directly addressed. However, you can obtain and alter the fields using the following instructions:

SETFPL Sets the PFP and FLI.

SETSFP Sets the SFP.

EXPS Exchanges the PFP and SFP values.

LDFP Loads the contents of the PFP into a register.

LDLN Loads the contents of the FLI into a register.

LDSFP Loads the contents of the SFP into a register.

LDSEGLN Loads the segment length into a register.

Private segments each have their own segment header. Shared segments have a separate headers for each logical work station to prevent one station's operation from interfering with another station's addressing.

Register Addressing

Register addressing means that all of the information required to address a field is contained in a register. You indicate register addressing by coding (reg) in instruction operands.

You can load a register with a register address using the LDRA instruction. In the following example assume that the displacement of FIELDA is 100.

```
FIELDA       DEFLD    S02,,6
             .
             .
             .
             LDRA     R04,FIELDA
*                                  load segment space ID,
*                                  segment number 2,
*                                  displacement 100 and
*                                  length 6 into register 4
```

Following the operation of an LDRA instruction the register will contain:

1. The segment space ID of the main storage segments assigned to this application program. This ID is always 1 for the segments allocated to the primary application program of each logical work station. The controller will assign new segment space IDs for secondary programs.

2. The segment number. This is the number (from 0 to 15) of a segment allocated to the logical work station. If the instruction refers to a segment that is not allocated, a program check occurs.

3. The displacement, in bytes, to a field from the beginning of the segment. This will be a value between 0 and 65 535. If a displacement is specified that is greater than the length of the segment, a program check occurs.

4. The length of the field, in bytes, determined by the length of field-2. If the displacement plus the length of the field is greater than the range of the segment, a program check occurs.

The displacement is in the low-order two bytes of the register. Therefore, you can change the displacement by adding or subtracting a value to or from the register. For example, to increase a displacement by 6:

```
SIX      DEFCON 6
         .
         .
         .
         ADDFLD R04,SIX
```

Modified Register Addressing

Modified register addressing means that an address value contained in a register is modified by another value. You must indicate modified register addressing by coding the (defrf) addressing form in the instruction operands.

(defrf) is the label of a DEFRF instruction that contains values used to modify an address in a register. A modified register address is formed as follows:

1. segment space ID - is taken from the register

2. segment number - is taken from the register

3. displacement - is obtained by adding the displacement from the register to the displacement of the DEFRF

4. length - is taken from the DEFRF.

For example, the following DEFRF associates register 3 with a displacement of 15 and a length of 5:

```
BALANCE   DEFRF R03,15,5
```

If register 3 contained the following address (perhaps initialized by an LDRA instruction):

```
segment space ID = 1

segment number = 4

displacement = 100

length = 200
```

then the following instruction:

```
ADDFLD R06,(BALANCE)
```

would add the *5-byte* field in segment space 1, Segment 4, displacement 115, to register 6.

The register (register 3 in the above example) is not altered when a modified register address is formed.

You can also use modified register addressing with DSECTS (Dummy Sections). You can define a dummy section by a series of DEFRF instructions. For example, assume a 20-byte customer record includes a 12-byte account number, a 3-byte status code, and a 5-byte current balance. This record could be defined as follows:

```
ACTNUM    DEFRF  R03,0,12
STATUS    DEFRF  R03,,3
BALANCE   DEFRF  R03,,5
```

Further, assume that Segment 4 contains a series of the 20-byte records described above, starting at displacement 100, and defined as follows:

```
RECORDS   DEFLD  S04,100,0
```

An application program could initialize register 3 to point to the first record by:

```
LDRA    R03,RECORDS
```

The displacement in register 3 would be 100 and the length would be zero. Now, if the program performs:

```
ADDFLD R06,(BALANCE)
```

then the balance from the first record (displacement 115, length 5) will be added to register 6.

The program could then add 20 to register 3 (the displacement part of a register address is in the low-order 2 bytes) as follows:

```
TWENTY   DEFCON 20
         .
         .
         .
         ADDFLD R03,TWENTY
```

so that register 3 would now point to the second record. The program could loop back to the ADDFLD instruction to add the second balance to register 6.

Programming Notes About Segment Headers

The purpose of this section is to give you some idea of the ways in which your program can manipulate and use the Segment Headers in the 4700. It concentrates on the SETFPL instruction which can be used to set the PFP and FLI for both fixed-length and variable-length fields. SETFPL has the following general format:

```
[label]     SETFPL     seg,pfp,fli
                        |   |   └an optional FLI operand
                        |   └an optional PFP operand
                       the segment number
```

Fixed-Length Fields

The simplest form of the SETFPL instruction refers to a DEFLD or DEFCON instruction used elsewhere in the program. The segment, displacement, and length values specified in the DEFLD, or implied in the DEFCON, are used by SETFPL to set the PFP and FLI.

The following example shows how DEFLD and SETFPL are used to set the PFP and FLI of a segment. After the SETFPL is executed, the PFP for Segment 2 is set to 0, and the FLI for Segment 2 is set to 100:

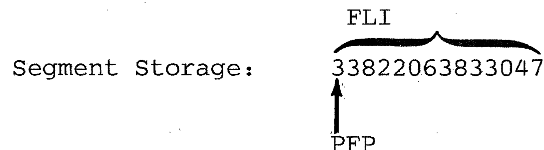
```
*
 OUTPUT            DEFLD     S02,0,100
 .
 .
 .
 SETPTRS           SETFPL    OUTPUT
```

The PFP and FLI may also be set using values in registers. The registers must be loaded prior to executing the SETFPL:

```
SETPTRS    SETFPL    S02,(R02),(R03)
```

Modifying the PFP and FLI: The PFP and FLI may be increased or decreased using the SETFPL instruction. This is done by using a signed number in the applicable operand.

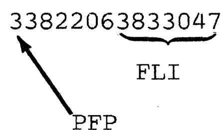
When the PFP is modified, an inverse modification is performed on the FLI. For example, if 5 is added to the PFP, 5 is subtracted from the FLI. After this operation takes place, the FLI is further modified by the FLI operand. For example, two 7-digit account numbers are in adjacent fields in Segment 2, and the PFP and FLI describe the first field:



The following SETFPL modifies the PFP and FLI to describe the second field:

```
INCRFLD    SETFPL      S02,+7,+7
```

When the PFP is increased by 7, the FLI is decreased by 7 to 0. Then the FLI is increased by 7. Thus, at the end of the operation, the second field is described:



Variable-Length Fields

You can define a variable-length field by inserting delimiters at the beginning and end of the field when it is created. The program can then use SETFPL to locate the delimiters and set the PFP and FLI accordingly.

You can also define a variable-length field as the data between the beginning of the segment and the first delimiter, or between the last delimiter and the end of the segment. You define delimiters themselves using the DEFDEL instruction described under -- Heading id 'delimit' unknown --. The rest of this section assumes that the appropriate delimiters have been defined and are recognized by SETFPL. The fields shown in the examples are typical of data entered by a teller using blanks, hex FA, and hex FB as delimiters. (Hex FA and hex FB have no EBCDIC character equivalent and are therefore represented by asterisks (*) in the text and figures.) The format of the fields is:

transaction accountno. amount amount EOM*

(EOM* is the end-of-message indication.)

The controller uses three operands—F, +F, and -F—when scanning a segment for variable-length field delimiters. When the controller finds the desired delimiter, it sets the PFP to the segment address of the first byte of the field and the FLI to the field length.

Note: Because scanning a field to set the FLI requires extra processing time, you should avoid this process, if possible, by setting the FLI to a valid absolute number. This stops the SETFPL instruction from scanning for the field length).

The following examples show the use of F, +F, and -F.

For example, the teller enters:

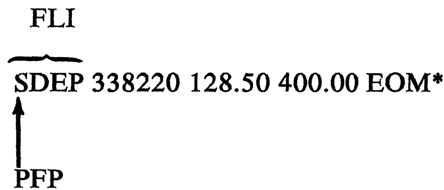
SDEP 3382206 128.50 400.00 EOM*

The entry is read into the input area, INPUTSEG, starting at the PFP. The read operation does not change the PFP or FLI.

To define the first field, you can code the following SETFPL:

```
FINDFLD        SETFPL INPUTSEG,F
```

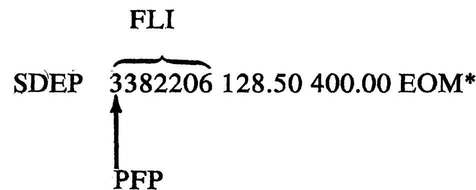
The pointers are now set as follows:



To point to the next field, you can code the following SETFPL:

```
FINDNEXT SETFPL INPUTSEG,+F
```

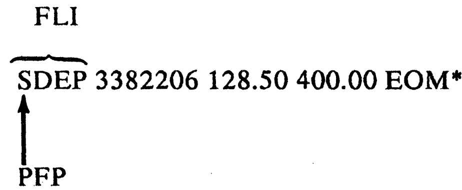
The pointers are now set as follows:



To point to the previous field, you can code the following SETFPL:

```
FINDLAST SETFPL INPUTSEG,-F
```

The pointers are now set as follows:



Programming Notes About Field Delimiters

To use delimiters you must choose a one-byte delimiter character, and then identify it using the DEFDEL instruction. Refer to the DEFDEL instruction description in "Data Definition Instructions."

Inserting Delimiters in Fields

The programmer should establish conventions for using delimiters. The following are suggested ways of placing delimiters at the beginning and end of the fields, depending on the source of the data:

Program-Created Fields: If only one delimiter is required, a routine can first fill the data area with the delimiter character, and then can place the individual fields into the data area leaving one space (delimiter) between fields. If more than one delimiter is required for the message, you can associate individual delimiters with various subfields.

Operator-Created Fields: The keys used by the operator to denote the end of a field or message must be set to the desired delimiters using the INTRTBL configuration instruction. An operator procedure must then be devised that describes how each field or message is ended (by pressing the space bar, an end-of-message key, or other convention).

Processing Messages and Fields

When a message is read by the application program, the message length is placed in Segment 1 at SMSIML, and the beginning of the message is identified by the PFP. Thus, the message is fully defined to the application program. However, the program must define individual message fields, either by using delimiters or by predefining fixed fields.

For example, the teller enters:

transaction accountno amount...EOM

If the beginning of the input area is known and the transaction type and account number have fixed lengths, these fields can be addressed as fixed fields. The remainder of the message contains a varying number of variable-length amounts separated by delimiters. Each is addressed as a variable-length field. To move the entire message, the FLI is set to the value in SMSIML.

The application program can also process transactions in which the teller enters the fields as individual messages. For example, the teller can press a key that indicates the type of transaction and EOM. The teller can then enter a field and press a key that indicates the account number and EOM. The teller can

continue to enter individual fields as discrete messages until the entire transaction is entered. As each field is read, it could be checked for errors and processed.

Logical Work Station Dispatching

The controller allocates processing time to, or *dispatches,* each logical work station either in work-station-ID-number order or according to a predefined table. This dispatch order, combined with other conditions that permit the stations to gain or release control, determines whether or not and when an application program runs on behalf of a logical work station.

The dispatched station and any others that have not completed processing by issuing an LEXIT instruction are *active.* All logical work stations that have never become active or that have indicated that processing is completed are *idle.*

The following are descriptions of how logical work stations gain and release control, or are *dispatched* by, the controller.

Dispatching Modes

The order and frequency with which work stations are checked for dispatching are determined by the type of dispatching mode in effect in the system:

- Station-Chain Dispatching—Stations are dispatched in station-ID sequence. No one station is checked for dispatching any more frequently than another.
- Priority Dispatching—You can choose to dispatch work stations in a certain order by defining the order in a priority dispatch table. The table, which you define during CPGEN, also controls how often a station is dispatched, compared to other stations. You can define more than one table and transfer control dynamically from one table to the other. More information on priority dispatching is under "Priority Dispatching."

Station chain dispatching is the basic dispatching mode. It is in effect unless a priority dispatching table was defined with the PRIDSP configuration macro and priority dispatching is invoked by the LCHAP (Change Priority) instruction. LCHAP also turns off priority dispatching.

In either mode, only those stations that have work to perform are dispatched; stations without work are passed by. One complete pass by the controller through the dispatch chain or priority dispatch table is called a *dispatching cycle.*

Priority Dispatching

Priority dispatching allows specification of the order and frequency with which logical work stations are checked for dispatching. An individual work-station ID can appear as often as desired in the table.

To use priority dispatching, at least one priority dispatching table must be defined in the controller configuration; the most you can have is four tables. A table is a list of station IDs in the order that they are to be checked for dispatching.

Entry Point Priority

It is possible for a station to have more than one type of work to perform when its turn occurs during a dispatching cycle. The controller, in this circumstance, gives control to the station's application program so that the type of work with the highest priority is performed. Work priorities are as follows:

1. Startup processing (occurs only once during a session)
2. Resumption of processing
3. Receiving data from a telecommunications line
4. Receiving data from a terminal
5. Receiving data from another station
6. Program interrupt
7. Asynchronous timer interrupt

Gaining Control

A work station can be dispatched when one of the following happens:

- The controller has just been loaded, the application program has a startup entry point (the STP operand of the BEGIN instruction), and startup has been specified for the station during configuration (the STARTUP operand of the STATION configuration statement). The station's application program is given control at the instruction label selected by the STP operand. A startup dispatch can occur only once during a processing session. (A *processing session* is the period of time between controller startup and restart or shutdown.)

- An active logical work station is ready to resume work that it began earlier. The logical work station may have given up control temporarily because of a data transmission operation. The station's application program is given control at the instruction pointed to by the station's instruction counter. A logical work station may be dispatched in this manner many times during the time it is active. Also, a logical work station may receive unsolicited messages while it is active (from the telecommunications line, a terminal, or another station). If the station does not read these messages before issuing LEXIT, it is redispatched; control is given to the application program at the appropriate entry point.

- The station receives a message from the telecommunications line, the station is idle, and the application program has an appropriate entry point (the ACP operand of the BEGIN instruction) entry point defined. The station's program receives control at the instruction selected by the ACP operand.

- The station received data terminal data, is idle, and the station's program defines a terminal entry point (the ATD operand of the BEGIN instruction). The program receives control at the instruction selected by the ATD operand.

- The station received data from another station, is idle, and the program defines a station entry point (the AST operand of the BEGIN instruction). The station's program receives control at the instruction selected by the AST operand.

- The station received a program interrupt (an LPOST issued by another program), the station is idle, and the station's program defines a program interrupt entry point (API operand of the BEGIN instruction). The station's program receives control at the instruction selected by the API operand.

- The station received an asynchronous timer interrupt, the station is idle, and the station's program defines asynchronous timer-entry point (ATM operand of the BEGIN instruction). The station's program receives control at the instruction selected by the ATM operand.

  The asynchronous timer gives an idle work station the ability to dispatch itself. This is in contrast to being dispatched because of an asynchronous request from another source. During a dispatching cycle, the controller compares the current value of the controller timer with a preset value in the station's program. The comparison result and the current station status (idle with no asynchronous requests pending) determine if the station is dispatched. The following conditions control timer operation:

  1. Any asynchronous requests pending when the timer request occurs, are processed first; the timer request is deferred.

  2. If the station is idle and the station's timer value is not zero but is equal to, or less than, the value of the controller timer, the station's program receives control at the entry point defined in the ATM operand of the BEGIN instruction. The station's timer value is reset to zero.

  3. Setting the station's timer value to zero cancels the timer request. The station's timer is not reset if the controller defers the timer request because another asynchronous request is pending. However, the station can change the timer value while processing a higher-priority asynchronous interrupt. Resetting the timer value to zero cancels any pending timer interrupt request.

In all cases except startup, a logical work station may be dispatched, at one of an application program's entry points, any number of times during an active session.

Releasing Control

Logical work stations give up control allowing other stations to be dispatched. Releasing control is done in the station's application program and may cause the logical work station to remain active or to become idle.

1. A logical work station remains active when it gives up control in one of the following ways:

   - A PAUSE instruction is executed. This causes the controller to take a dispatching cycle. Control will be returned to this station's program at the next sequential instruction.

- A data transmission instruction is executed that requires a pause in processing until the operation is completed. Other logical work stations are checked and may be dispatched. Control is returned to the program that released control at the next sequential instruction.

- An LWAIT instruction is executed. Other logical work stations are checked and may be dispatched. Control is returned to the next sequential instruction when an asynchronous interrupt occurs or an operator signals an attention.

- A program check causes control to be given to a program check routine (the PC operand of the BEGIN instruction).

2. A logical work station becomes idle when it gives up control in one of the following ways:

   - An LEXIT instruction is executed.

   - A program check occurs that causes an LEXIT.

System COPY Files

Segments 0, 1, 14, and 15 contain fields that have special meaning during controller application program execution. Definitions of these fields are made available by coding the following operands of the COPY instruction:

- DEFRGS or DEFREG, which defines the registers in Segment 0

- DEFSMS, which defines the fields in Segment 1, the System Machine Segment

- DEFAPB, which defines the fields in Segment 14, the application program header

- DEFGMS, which defines the fields in the fixed portion of Segment 15, the Global Machine Segment.

Because the definitions are subject to change, you should base any references to the standard definitions on the labels provided by the system copy files. You should also ensure that fields defined by a system copy file are referenced individually.

For example, COPY DEFAPB might contain:

```
APBLBL1  DEFLD  S14,,2
APBLBL2  DEFLD  S14,,2
```

Because it is possible that the DEFAPB definition could change, it would not be advisable for you to code one DEFLD (of length 4) and expect it to contain both fields.

Appendix B contains details of system copy files. Those copy files that contain the DEFxxx notation can be created in two forms; one for the Segment-Displacement addressing and the other for modified register addressing. For example, DEFCPL will expand into a series of DEFRF instructions for modified register addressing, if the following is used:

```
         LDSECT BASE=r
         COPY   DEFCPL
         LEND
```

```
where: r is a register number (0 - 15)
```

DEFCPL will create a series of DEFLD instructions, for Segment-Displacement addressing if you code an EQUATE before the COPY instruction, as follows:

```
DEFCPLS  EQUATE s
         COPY   DEFCPL
```

where: s is a valid segment number

Note:

Appropriate volumes of this *4700 Controller Programming Library* contain similar DEFxxx COPY parameter lists and files. Refer to the volume for the type of programming you are performing, for detailed descriptions of the applicable DEFxxx files.

Condition and Program Check Codes

Condition and program check codes supply information about program operation and any errors that might occur:

- Condition codes: Some instructions set one or more condition codes. Condition codes indicate the result of the operation of an instruction. Each instruction description in Chapter 5 includes the condition codes that can be set by that instruction.

- Program check codes: Program checks are indications of errors detected by the controller during operation of an application program. In general, they indicate that the application program has attempted to instruct the controller to perform an invalid operation. For example, scan a field for a delimiter without providing the required delimiter table.

  Your application program can include a routine to handle program checks. Its entry point is defined by the PC operand of the BEGIN instruction. When the program check routine receives control, a flag (SMSPCR) is set to indicate that a program check routine is in control. If this flag is not reset by the application program, it is reset when an LEXIT is issued.

  All program checks cause the controller to write a message to the system log that contains the station number, application program name, program check code, program check address, loop instruction count, and the first two bytes of the failing instruction. In addition, the program check code and program check address of the failing instruction are placed in the System Machine Segment (Segment 1). All of this information can be printed on a terminal or transmitted to the host processor by a program check routine. A program check can also appear as a CNM (Communications Network Management) alert if specified in your configuration.

The controller uses the following rules when a program check occurs, depending on whether a program check processing routine is active:

- If SMSPCR is off, give control to the program check entry point of the current application program.
  - If there is no program check entry point and the current program is the primary program, then perform an LEXIT.
  - If there is no program check entry point and a current application is the secondary program, then perform an APRETURN; place an entry in the system log showing that an APCALL caused a program check; and give control to the program check entry point of the calling program. If the calling program does not have a program check entry point, the controller will continue to perform APRETURNs until a program check entry point is found or an LEXIT is performed because the primary application has no program check entry point.
- If SMSPCR is on, the controller will attempt to give control to a program check entry point as above, *except* that it will *not* give control to the program check entry point of the current application program.

Optional Instructions

4700 subsystem controller instructions are executed by controller modules incorporated into the configuration image. For most 4700 instructions, the controller includes the required modules in the configuration image during its generation. However, for certain instructions, the controller does not automatically include these modules in the configuration image; they must be specifically requested in the specifications you use to define the configuration image.

Including optional instruction modules will affect the amount of storage required for controller functions and may create a need for additional storage.

Even though the appropriate controller data modules are included in the configuration image, the programmer should be aware that the control operator can prevent these modules from being loaded during the controller load procedure.

If the appropriate controller data modules are not in the controller when an optional operation code is encountered, a program check hex 09, invalid operation code, occurs.

The Global Machine Segment (GMS) contains information that can be tested for the presence of optional modules. See the DEFGMS copy file in Appendix B.

Volume 6: Control Program Generation of this 4700 Programming Library contains the information you might need on this subject.

COBOL Considerations

4700 assembler programs that transfer control to and from COBOL programs require additional instructions if they receive or send parameters, and must follow these conventions:

1. A called assembler program receives an APCALL parameter list from the calling COBOL program set as follows:

   - Shared segment flags = X'BFFA'
   - Register flags = X'4000' (Passes register 1)

2. Register 1 contains a register address of a list of addresses, each of which selects a parameter. The register-1 length field defines the address-list length.

3. The contents of Register 12 are destroyed by COBLCALL.

The called assembler program should have an FCLENTER instruction at the APCALL entry point to define a save area for the contents of register 1 and assign names to the parameters. It should also have an FCLEXIT instruction at the APRETURN point to restore the contents of register 1 and redefine the parameters to be returned to the COBOL program.

The 4700 COBOL Compiler generates 4700 assembler output that can contain two additional instructions, CRETN and LCONVERT, primarily for COBOL use. See the *IBM 4700 Finance Communication System, COBOL Programmer's Guide*, SL23-0082, for more information.

Use of 3600 Programs

The 4700 system accepts 3600 application programs with little or no change. Application programs that use non-standard forms of SMS (the System Machine Segment) or GMS (the Global Machine Segment) may require changes to operate with different 4700 system formats.

4700 allocations for the global (GMS), system (SMS), and application program storage segments are larger than for the 3600 system. Those 3600 programs that refer to the GMS and SMS fields defined by COPY DEFGMS or COPY DEFSMS are not affected by the larger GMS and SMS segments in the 4700. Those programs that use other than the standard system definitions may require redefining of those areas. Also, those configurations at or near the limits of their segment specifications for the GMS, SMS, or Segment 14 might need testing to ensure that enough storage remains to accommodate the larger segments.

There have been a number of changes in 4700 Control Program Generation. You may find it necessary to modify your 3600 CPGEN specifications for them to be acceptable in the 4700 environment.

Chapter 3. 4700 Instruction Categories

This chapter introduces to you various kinds of 4700 instructions. They are divided here into categories to help you understand their purpose and function. *Not* all of the controller's base instructions are included, and this chapter does *not* contain detailed information. Chapter 5 contains descriptions of each instruction including examples.

Program Definition Instructions

This section describes the instructions that are used to control assembly of application programs and program sections.

You must place an APOPT (Application Program Options) as the first instruction in an assembly. BEGIN and FINISH are used to define an assembly. SEGCODE (Application Program Section Identifier) and ENDSEG (End Application Program Section) are used to define relocatable sections that may be appended to relocatable root sections and overlay sections.

Assembly Definition

The APOPT Instruction: APOPT precedes all other 4700 assembler instructions in an assembly. It is used to specify whether the instructions that follow are part of a nonrelocatable program (RELOC=N or an omitted operand) or a relocatable program (RELOC=Y). If the APOPT is omitted, a nonrelocatable assembly is performed. APOPT may be specified only once in an application source program.

APOPT also has other functions:

- DISP=NEW specifies that the section or sections are new and should only be added to the host library if a section with the same name does not already exist.

- DISP=OLD (or an omitted operand) specifies that a section or sections may already exist in the host library. If the same name(s) already exists, then the section(s) being assembled replaces those in the library. If the section does not exist, it will be added.

- DIRNAME=(name,NEW) specifies that the named directory should be created in the host library and that directory entries for all sections in the assembly should be added to the new directory. If a directory with the same name already exists, another directory will *not* be created and the entries for the sections will *not* be added to the old directory.

- DIRNAME=(name,OLD) specifies that the named directory already exists and that directory entries for all sections in the assembly should be added. (An entry will not be added if a duplicate entry already exists in the directory.) If the directory does not exist, a new one is created and entries for all sections in the assembly are added to it.

- YL2=Y specifies that all AL2 address constants used as operands of DEFCON instructions in the following sections should be changed to relocatable YL2 address constants. You should be careful when using this operand because the DEFCON macro cannot distinguish between expressions that are not intended to be relocated and those that are. You should

therefore manually change the source statements if there is any question about the use of AL2 address constants in the application program. If YL2=N is specified or the operand is omitted, AL2 address constants will not be altered.

The BEGIN Instruction: You use the BEGIN instruction to identify the application program and specify its entry points. It may also be used to specify the address of a delimiter table.

The application program name consists of an eight-character name (names are padded to the right with blanks), a date, and a version number. When the application program is processed by Host Support, the name and version number are used to associate the program with a load image. The name is also used to associate an application program with a station. Application programs with the same name cannot be in a controller at the same time.

Program entry points define where processing begins for:

initial startup The STP entry point is specified if the application program is to gain control when the controller is loaded by the control operator.

data received asynchronously Three entry points are used when data is sent to the program running on behalf of a logical work station:

- AST for data from another station
- ATD for data from a terminal
- ACP for data from the central (host) processor.

entry from another program The APENTRY entry point must be specified if your program can be called by another.

interrupts from another program. The API entry point must be specified if your program can be interrupted by another.

timer functions The ATM entry point must be specified if the station timer facility is to be used to dispatch your program.

program checks The PC entry point must be specified if your program contains a routine to handle program checks.

The following conditions occur when an entry point is not specified and an attempt is made to dispatch the application program at that entry point:

- With no STP entry point and startup specified for the logical work station: a program check.
- With no AST, ACP, ATD, ATM, or API entry point the dispatch will be suppressed and held. With APCALL/APRETURN the entry point may be defined in another application program's BEGIN instruction.
- With no APENTRY point and an APCALL issued by another station: a program check.

The FINISH Instruction: FINISH defines the end of the assembly and must be the last instruction in the assembly.

Section Definition

The next few paragraphs introduce the instructions that you can use to define different kinds of program sections. Detailed information about these instructions is in Chapter 5.

The SEGCODE Instruction: SEGCODE must be the first instruction in a relocatable section that will be appended to another assembly during link-edit. It specifies the name and version number of that section.

The ENDSEG Instruction: ENDSEG defines the end of a section begun by a SEGCODE instruction.

The OVLYSEC Instruction: The OVLYSEC instruction must be the first instruction in overlay section.

The ENDOVLY Instruction: The ENDOVLY instruction defines the end of an overlay section begun by an OVLYSEC instruction.

The SECTION Instruction: SECTION is used to define a dummy section (DSECT). A dummy section is a description of an area of storage that is defined elsewhere in the program or in another assembly. It may appear anywhere between the BEGIN/FINISH, OVLYSEC/ENDOVLY, or SEGCODE/ENDSEG instruction pairs and may appear as many times as are required. A SECTION DUMMY must be ended by a SECTION END.

Assembly Control Instructions

Equates

Using an absolute value in symbolic instructions has two disadvantages:

- If a value such as a register number is changed, many instructions may have to be changed.
- Absolute numbers do not convey meaning and make it difficult to understand the logic of a program when reading the listing.

The EQUATE instruction allows you to associate a meaningful label with an absolute value; the absolute value must still conform to the specifications of the operand. It also defines information for the assembler and makes your program easier to modify.

COPY Instruction

Use of the COPY instruction is described in Chapter 2.

Controlling Base Registers during Assembly

USEBASE, SAVEBASE, and REBASE allow you to control the base register numbers that are assembled into instructions that use modified register addressing

with the LDSECT instruction. The USEBASE instruction must refer to an LDSECT instruction that describes an area to be addressed by a register. LDSECT is one of the Data Definition instructions in this chapter.

Assembly Listing Control Instructions

This section mentions the instructions that are provided so that you can control the printed output of the assembly process. They are: LEJECT (Eject a Page); LSPACE (Space a Line); PLPCMD (Post List Processor Command); and PRINTI (Print Macro Expansions).

These instructions allow you to leave blank space in your assembly listing to improve its readability. They also allow you to have some control over what is printed and what is not printed.

Data Definition Instructions

This section describes the instructions that define data, both in the form of constants and in the form of areas to be used for input, intermediate, or output storage.

Defining Constants

The instruction that defines a constant field is DEFCON (Define Constant). DEFCON is used to create bytes of data within the application program. This data can be used, for example, as prompting messages for the teller. When a DEFCON is assembled, it has an implied:

segment number (14)

displacement (its location in the application program)

length (the number of bytes defined).

Defining Delimiters

The DEFDEL (Define Delimiters) instruction defines delimiter characters that can be used to process variable-length fields. Use of delimiters is discussed in Chapter 2 under "Programming Notes About Field Delimiters" on page 2-22.

Defining Dump Parameters

The DEFDMP (Define Dump Constants) instruction is used in conjunction with the APBDUMP instruction to request that the station dump one or more segments to a diskette.

Defining Masks and Modulus Factors

The MASK and MOD (Modulus) instructions create constants that are used during the execution of the EDIT and MODCHK (Modulus Check) instructions respectively.

Defining Tables

The LTRTBEG, LTRTENT, and LTRTGEN (Translate Table Begin, Entry, and Generate) instructions create a table that is used by the LTRT (Translate)

instruction. The TABLE instruction creates a table that is used by the LSEEK and LSEEKP instructions.

Three instructions, LTRTBEG, LTRTENT, and LTRTGEN, are used to define and generate a translation table during assembly of the application program. This translation table is used by the LTRT instruction. LTRTBEG and LTRTENT define the characteristics and contents of the translation table; LTRTGEN generates the table.

Defining Fields

DEFLD (*define field*) is used to associate a label (symbolic location) with a field definition. Using DEFLD instructions makes coding easier because the field so defined can be referred to symbolically.

The DEFRF (define modified register-addressed field), LDSECT (dummy section), Ln (DSECT level definition), and LEND (DSECT End) instructions are used to describe areas of segment storage that can be processed using modified register addressing.

Data Operation Instructions

This section describes the instructions you can use to perform: general data movement; translation; editing; and a large number of other operations.

Formatting Input Data

For the convenience of an operator, it is usually desirable to allow monetary values to be entered from a keyboard in a relatively free form. However, these free-form values frequently must be reformatted by the application program for the values to be suitable for arithmetic processing. The SCALE instruction, together with a scale parameter list (see COPY DEFSCA instruction), can be used to perform such operations. In addition, the SCALE instruction provides validity checking on the input data.

Moving Data within Controller Storage

The Move and Convert Zoned (MVCZ); Move Field (MVFLD) and Move Field Reverse (MVFLDR); Move Fixed (MVFXD) and Move Fixed Reverse (MVFXDR); and Move Segment (MVSEG) and Move Segment Reverse (MVSEGR) instructions move data from one storage location to another. Movement can either be between segments or between locations in the same segment.

Move Data Immediate (MVDI) moves one or two bytes of immediate data. After executing any of these instructions, the PFP of the result segment points one byte past the result field. Successive operations for moving data to the receiving segment can be performed without resetting the PFP, unless register or modified register addressing is specified.

Verifying Data

The VERIFY Instruction: VERIFY can be used to check a field for data type and length. The result of the check is indicated by one or more condition codes. The application program tests the condition code to determine whether the field is valid or invalid.

For example, all savings account numbers are seven numeric digits. The VERIFY instruction allows you to test an account number to determine that it has 7 characters and that they are numeric.

The MODCHK Instruction: Modulus checking can also be used to help determine that an operator has not entered an account number incorrectly such as: reversing two digits, entering a non-numeric EBCDIC character, or entering too many characters. For example, when the account number is initially generated, the first six digits could be arbitrarily selected and the seventh digit could be calculated so that the account number will pass the modulus check.

Data Translation

The LTRT instruction translates 8-bit input codes as specified by a translation table within the application program. The location of the translation table and other information needed for LTRT execution are contained in a parameter list. The parameter list must be initialized before LTRT is executed. (The parameter list fields are defined by the COPY DEFTRP instruction.) After operation of LTRT, some of the parameter list fields will describe the results of the LTRT operation.

The translation table (there may be 4) is defined and generated during assembly of the application program (using the LTRTBEG, LTRTENT, and LTRTGEN instructions). An input area into which the input code stream is read and an output area to contain the translated output from LTRT are required, and are pointed to by the parameter list.

Each input code may have translation, translation control, and program control definitions assigned to it in the translation table. These definitions control LTRT and program operation as follows:

1. Translation Definition: The input code may be translated into 1 to 7 characters of output, or defined as a character for which no output is to be generated.

2. Translation Control Definition: The input code can be defined to cause a shift from one translation table to another; to permit input codes to be passed to the output area without processing (transparent write); and to control the position of the next character in the output area (backspaced or advanced). Positioning does not destroy characters already in the output area. The input code can also be defined to cause a user function code to be entered in the LTRT parameter list when execution ends on the associated input code.

3. Program Control Definition: The input code can be defined to end LTRT execution with program control passing to the next sequential instruction in the application program or to a specified address in the application program.

Any input code may be assigned any combination of the three definitions just noted. If none of the definitions is specified, the input character will be treated as a null character, with no corresponding translation output or control function associated with it. If more than one definition is specified, the operations are performed in the following order: translation, translation control, program control.

Table Lookup

The LSEEK and LSEEKP instructions allow you to compare a field with elements in a table. They can be specified so that when a match is found between the field and a table element, a branch is taken to an address associated with either the table element or the instruction. LSEEK searches a table sequentially.

LSEEKP searches a table using either a sequential search or a binary search on a sorted table.

Tables can be created in one of two ways:

- when the application program is assembled by specifying a TABLE instruction or using one or more DEFCON instructions
- during application program execution.

A table, such as table of savings accounts that require special processing, can also be built by the controller application program. The host application program in the host processor can send a list of account numbers daily. The controller application program can then build the table in one of the global segments so that all logical work stations have access to the table.

Packing and Unpacking Data

The four instructions for packing and unpacking hexadecimal data (0-9 and A-F) are: PAKFLD, PAKSEG, UPKFLD, and UPKSEG. These instructions convert a byte into 4 bits or convert 4 bits into the EBCDIC hexadecimal equivalent. Data compaction using these instructions requires less execution time than the load-and-convert and store-and-convert instructions (which convert decimal numbers to binary and binary numbers to decimal).

Packing Instructions

You can use PAKFLD and PAKSEG to convert a byte of EBCDIC hexadecimal data (X'F1'-X'F9' and X'C1'-X'C6') into the 4-bit binary equivalent. The data is packed from left to right; the resulting field occupies half the storage of the original field. If an odd number of digits is converted, the leftmost 4 bits of the resulting field are set to 0. If digits other than 0-9 and A-F are packed, the results are unpredictable.

Unpacking Instructions

You can use UPKFLD and UPKSEG to convert any 4 bits into the EBCDIC hexadecimal equivalent. The resulting field requires twice the space of the source field. The same procedure is used to unpack data as to pack it.

For example, the application program could contain a routine that converts the binary number to printable form and displays it on the teller's display. UPKFLD or UPKSEG could be used to convert X'A200' to X'C1F2F0F0' (a printable A200). Unpacking is from right to left, and the fields can overlap if result (unpacked) data does not replace packed source data.

Compression and Compaction

You can use the COMP instruction to reduce the size of a data stream to be sent to another system or to be stored on an auxiliary storage device. Compression replaces a string of from 3 to 63 *duplicate* (repeated) characters with a 1- or 2-byte code. Compaction replaces *pairs* of commonly used characters with single coded bytes. Normally, compression is used when a data stream contains long sequences of identical characters: blanks, zeros, or nulls, for example. Use compaction for character pairs that are repeated frequently, such as: th, sh, ea. (You would not normally compact uppercase characters unless your data streams use uppercase only.)

Data Compression

Compression replaces strings of repeated prime characters with a 1-byte compression code. The *prime character* should be the character most frequently repeated in the data stream. (Normally, the prime character used is the blank or space hex 40 or the null hex 00.) However, compression also compresses repetitions of other characters, called non-prime characters. Each string of repeated non-prime characters is compressed into 2 bytes (a 1-byte code followed by the character being compressed). For example, if a data stream contains:

$$$$$$$$aaaaa$$$$$bbbbbb

and the prime character selected is the dollar sign ($), the compressed string will contain:

- 1 byte indicating 8 repetitions of the prime character.
- 2 bytes indicating 5 iterations of the "a".
- 1 byte indicating 5 iterations of the prime character.
- 2 bytes indicating 6 iterations of the "b".

The actual compressed data stream will contain these codes in the form of string control characters (SCB). See "String Control Characters" in this chapter for a description of the format of a compressed data stream.

Data Compaction

Compaction reduces frequently used *character pairs* to 1-byte compaction codes. For example, if the data stream contains many occurrences of the character pair "th", the t and the h would be defined as *master characters.* During compaction, each occurrence of the pair "th" would be reduced to a single byte. All other characters would remain as 1-byte characters (but not in their normal EBCDIC notation). The number of master characters (maximum is 16) is governed by the total number of unique characters that could occur in the data stream (the compaction set).

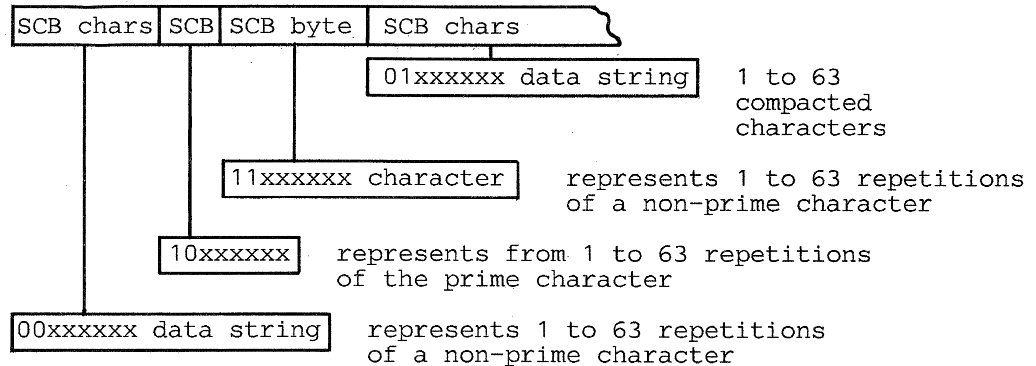
After you use COMPTB to build the compaction table and before the COMP instruction is issued, the location of the compaction table is placed in the CPLTBS and CPLTBD fields in the COMP parameter list (DEFCPL). On execution of COMP, each *pair* of master characters in the data stream is compacted by a 1-byte code. Any non-master character, and master character *not paired* with another master character, occupies a full byte in the compacted data stream, but not in its normal EBCDIC format. (The EBCDIC characters are encoded to the compaction table values.) Only *paired* master characters are compacted.

String Control Characters

The compression and compaction methods used are the standard Systems Network Architecture (SNA) compression and compaction procedures. The output data stream is a series of smaller data strings separated by codes called *string control characters.* The string control character identifies the data string that follows it, and describes the length of that string. After compression, for example, the output data stream will contain a series of data strings in compressed mode. Each string is preceded by a string control character (SCB). Each SCB begins with a 2-bit code describing the kind of string, and a 6-bit field containing the length of the string.

SCB Code:	Data String Described:
00xxxxxx	This SCB describes a string of from 1 to 63 uncompressed and uncompacted characters; the data is unchanged from the input data stream. The xxxxxx bits contain the length of the data string, identifying the location of the next following SCB. The actual data string of from 1 to 63 characters immediately follows this SCB.
01xxxxxx	This SCB describes a string of from 1 to 63 compacted characters in the following data string. The xxxxxx bits contain the length of the following string. Note that a string of compacted characters can contain: single bytes representing pairs of compacted master characters; single bytes containing single, unmatched master characters; and single bytes containing the compact code for non-master characters.
10xxxxxx	This SCB describes and *replaces* a string of from 1 to 63 consecutive *prime*, compressed characters from the input data stream. The xxxxxx contains the number of consecutive prime characters represented by this SCB. No data follows this SCB.
11xxxxxx	This SCB describes a string of from 1 to 63 consecutive non-prime, compressed characters from the input data stream. The xxxxxx bits contain the number of times the characters are repeated in the input data stream. The byte immediately following this SCB contains the non-prime character being repeated.

After compression and compaction have been performed on an input data stream, the format of the output data stream (and its data strings) might look like this:



Data Decompression

The DECOMP instruction decompresses a data stream based on the parameter information contained in the list described by the COPY DEFDCP instruction. (This list must not be in Segment 14.) During decompression, the input data stream containing compressed characters is restored to its decompressed state in the output area. Fields in the DEFDCP parameter list are set to indicate return information. If a data stream has been compressed for transmission to another program at the central computer, for example, the receiving program must be given (using standard SNA protocols) the prime compression characters.

Data Decompaction

The DECOMP instruction decompacts a compacted data stream using both a DEFDCP parameter list and a decompaction table created with the DECOMPTB instruction. To permit decompaction and decompression processing, set the DCPFCT flag on and store the table address in DCPTBS and DCPTBD.

If a data stream has been compacted for transmission to another program, at the central computer for example, the receiving program must have a copy of the decompaction table.

Arithmetic and Logical Instructions

This section describes the two types of arithmetic operations performed by the 4700, binary and zoned decimal, and the logical (AND, OR, Exclusive-OR, and comparison) operations that can be executed.

Arithmetic Operations

The 4700 controller performs arithmetic in either binary or zoned decimal. The EBCDIC data entered by the terminal operator can be operated on either directly by the zoned decimal instructions. EBCDIC data can also be converted to binary by the application program before an arithmetic operation and the results reconverted back to EBCDIC form for other use such as printing or displaying.

Binary Operations

LDFLD and LDSEG are used to load a binary number into a register. LDREG replaces the contents of one register with the contents of another register. STFLD and STSEG are used to store the contents of a register into a field. The data is loaded or stored without conversion. The LDFLD and LDSEG instructions are used when a binary number is generated in the application program or a value is retained in binary form. If the number being loaded is shorter than 6 bytes, the leftmost bit is propagated to the left in the register. If the field into which the number is stored is shorter than 6 bytes, the number being stored is truncated on the left.

LDDI loads 2 bytes of immediate data from the instruction into the low-order 2 bytes of a register; the high-order 4 bytes are set to zeros.

LDFLD, LDSEG, LDDI, STFLD, and STSEG do not change the PFP or FLI during execution; other instructions must be issued to set the PFP and FLI. The following instructions store the rightmost 4 bytes of a register into a field:

```
SETFPL    OUTSEG,,4     Sets the FLI for amount of data.
STSEG     1,OUTSEG      Stores the rightmost 4 bytes.
```

LDFLDC and LDSEGC convert an EBCDIC decimal number (hex FO-hex F9), which may be preceded by a minus sign (hex 60) into a 6-byte binary number, and load it into the specified register. A negative number is loaded in twos complement form.

The 4700 Assembler Language contains instructions that perform the following arithmetic operations.

- Addition: ADDREG and ADDFLD
- Subtraction: SUBREG and SUBFLD
- Multiplication: MPYREG and MPYFLD
- Division: DIVREG and DIVFLD

STSEGC and STFLDC convert the binary contents of a register into an EBCDIC decimal number (FO-F9) and store the result in a specified field. The largest possible decimal number resulting from a store-and-convert instruction is 15 digits. However, if the number will never exceed 10 digits, the field can be specified as being 10 bytes long, and truncation, which occurs to the left, can be ignored. When the field is stored, a condition code is set indicating a positive or negative number so that the proper sign can be added.

Zoned Decimal Operations

Zoned decimal operations do not require converted operands, but do require sign indicator processing by the application program. Both operands and results contain a sign indicator in the high-order four bits of the least significant digit. The application program must set a negative indicator in those bits on negative operands. The program must also analyze the same bits of a zoned decimal result to determine the sign, and then format any output accordingly. In other words, the sign of an operand or result is not automatically changed to an appropriate displayable character.

The following are the zoned decimal instructions, and the functions they perform:

- ADDZ—Add Zoned Decimal
- COMPZ—Compare Zoned Decimal
- DIVZ—Divide Zoned Decimal
- MPYZ—Multiply Zoned Decimal
- SUBZ—Subtract Zoned Decimal

The zoned decimal instructions operate on operands that are character strings in a storage segment. The result, where created, replaces the first operand. All operands have the standard zoned decimal form—each byte is a digit with X'F' in the high-order four bits and a digit (X'0'—X'9') in the low-order four bits. The sign position, which the application program must set, contains a positive (+) sign code (X'A', X'C', or X'F') or a negative (–) sign code (X'B', X'D', X'E'). All zoned decimal operands are processed byte-by-byte, from right to left. The maximum operand length is 63 bytes, but operands greater than 15 bytes must be specified with register addressing or modified register addressing.

Comparisons

The 4700 has instructions that compare the contents of two fields, a register and a field, two registers, or a field and immediate data:

- CAFLD and CAREG perform a binary arithmetic comparison. The leftmost bit of each field or register is checked to determine whether the number is positive (bit is 0) or negative (bit is 1), and the numbers are compared to determine which is larger.

- CCFLD, CCFXD, and CCSEG perform a logic comparison of the data in two fields. CCDI logically compares the contents of a fixed-field with a 1- or 2-byte immediate operand in the instruction. The comparison is performed on the EBCDIC representations: an A is less than a C (C1 is less than C3), and F is greater than a $ (C6 is greater than 5B).

- COMPZ performs an algebraic comparison of two zoned decimal operands. A shorter operand is logically padded to the left (high-order positions) with (X'FO').

If your application program is going to perform multiple comparisons to find equivalent fields, the LSEEK instruction should be used.

Logical Operations

AND and ANDI perform logical AND operations, INOR and INORI perform inclusive OR operations, and EXOR and EXORI perform exclusive OR operations. They operate on two fields or on a field and 1 or 2 bytes of data in the instruction (immediate data).

AND and ANDI

An AND operation produces the logical product of the bits in two fields. If both bits in the same relative position are 1's, a 1 is set in the result. Otherwise, a 0 is set in the result.

Field 1	01100011
Field 2	11001011
Result	01000011

INOR and INORI

An inclusive OR operation produces the logical sum of the bits in two fields. If either bit in the same relative position is a 1, a 1 is set in the result. If both bits are 0, a 0 is set in the result.

Field 1	01100011
Field 2	11001011
Result	11101011

EXOR and EXORI

An exclusive OR operation produces the modulo-two sum of the bits in two fields. If only one bit is a 1, a 1 is set in the result. Otherwise, a 0 is set in the result.

Field 1	01100011
Field 2	11001011
Result	10101000

Testing Bits

LIFON and LIFOFF test a single bit, TSTMSK and TSTMSKI test from 1 to 16 contiguous bits in a 1-byte or 2-byte field. The mask specified in the instruction determines which bits are tested; a mask bit set to 1 indicates that the corresponding data bit is to be tested. A condition code indicates the result of the test.

Setting and Resetting Bits

You can set a bit on by using the INOR, INORI, or LSETON instructions. You can reset a bit (set it off) by using the AND, ANDI, or LSETOFF instructions.

Shifting Data

SHIFTL and SHIFTR are used to shift the contents of a register left or right. Any significant bits that are shifted out of the register are lost. As the bits are shifted, zeros are inserted as padding. A maximum of 16 bits may be shifted for each execution of this instruction.

Shifting a number in a register to the left has the effect of multiplying the register contents by powers of 2. Shifting a positive number in a register to the right has the effect of dividing the register by powers of 2.

Program Control Instructions

This section describes the instructions that you will use to: determine the sequence in which your program will be executed; to invoke other programs; to transfer control to various parts of your program; and to execute single instructions outside of the sequence of execution.

Call Programming Instructions

The APCALL and COBLCALL instructions are the ones you will use to invoke another assembler language application program or a COBOL application.

The FCLENTER instruction is used to receive control and parameters from a COBOL program.

You will use the APRETURN and FCLEXIT instructions to return control to an assembler application or to a COBOL program that called your program.

Passing Data Between Programs

You can pass data such as operands and addresses between programs by allowing the programs to share the same segment storage or by placing the address of the data in a register. In the cases where storage cannot be shared, you can have the calling program load the address of an operand into a register with LDRA, and the secondary program refer to the field using a DEFRF instruction label that refers to that register address.

Your assembler language program can use the six-byte registers in segment 0 to pass addresses or other values from your program to a called assembler program. You must define the specific registers, any segments your program will share with the called program, and the called program name in a parameter list selected by APCALL. If your program receives parameters from the called program, you should either share a segment with the called program that it can use to return data, or establish an area selected by a register address that the called program can use to return parameters to you.

Instructions that Release Control

You may use the LEXIT, LWAIT and PAUSE instructions to discontinue execution of your program for a short time or until something happens that should cause your program to continue.

Branch Instructions

Branching instructions in the 4700 system fall into four categories: those that test a condition code set by a previous instruction, those that test a bit switch (on or off), those that test an index value, and those that link to a subroutine.

Branch on Condition Code Instructions

JUMP, BRAN, BRANL, BRANR, and BRANLR test the condition code set by a previous instruction and change the sequence of program execution if the tested condition exists. Each bit of the testing value in the branch instruction (the *mask*

operand) corresponds to a bit in the rightmost half of the condition code byte in Segment 1 (at SMSCCD). The branch mask is set using the rightmost 4 bits specified in the mask byte when the instruction is coded; for example, if X'0F' is coded, the branch mask is set to X'F'. Setting a mask bit to 1 tests the corresponding condition code bit. If the branch mask is X'F', the branch is always taken (an unconditional branch); if the branch mask is X'0', the branch is never taken (no operation). If the mask is not specified in the instruction, the mask is set to X'F'.

The branch mask must be set by using a mnemonic; by specifying a hexadecimal value; or by specifying the label of an EQUATE instruction. If an EQUATE is used, its label should not duplicate any of the mask mnemonics that are defined by the 4700 Controller.

The JUMP Instruction: JUMP is a branch instruction that:

- Requires only 2 bytes of storage.
- Is faster than other branches.
- Has a maximum range of 510 bytes (255 halfwords).

The branch-to address is generated during assembly.

The BRAN Instruction: BRAN assembles into a 4-byte instruction that has a range of 64K bytes. The branch-to address is generated during assembly.

The BRANR Instruction: BRANR assembles into a 2-byte instruction. The application program must place the branch-to address in the rightmost 2 bytes of a register before BRANR is executed.

Branch on Bit Switch Instructions

The LIFON and LIFOFF branch instructions are used to control program flow by testing a bit (a bit switch) and changing the sequence of program execution if the tested switch is on (LIFON) or off (LIFOFF). If the setting that is tested is not found, program execution continues with the next sequential instruction. The instructions operate on a 2-byte field, and any single bit in the field may be tested. The bits (bit switches) are numbered 0 to 15, starting with the high-order bit in the field.

LIFON and LIFOFF have the ability to conditionally set the bit switch being tested. LIFON can set the switch on if it is off, and LIFOFF can set the switch off if it is on. The branch is not taken in these cases.

Two bit-setting instructions, LSETON and LSETOFF, may be used to unconditionally set the bit switches tested by the branch instructions. Both instructions operate on the two-byte field used by LIFON and LIFOFF. When hexadecimal notation is used to specify the bit switches to be set, more than one switch may be set with a single instruction execution; for example, specifying x'C000' in the instruction would set bit switches 0 and 1 on or off as desired.

Branch on Index Instruction

BRANX provides index increment, compare, and branch abilities in a single instruction. This instruction can be used to control the number of times a series of instructions is executed. The branch-to address is specified in the instruction along with a register that is initialized before BRANX is executed. This register contains three 2-byte fields: the comparand, the increment, and the index value. The increment is added algebraically to the index value and the sum is placed in the index field. The updated index field is then compared with the comparand. If the comparison is unequal, the branch is taken. If the comparison is equal, the branch is not taken and processing proceeds with the next sequential instruction.

Branch and Link Instructions

BRANL and BRANLR store the address of the next sequential instruction if the branch is taken. These instructions provide a method of using subroutines. The difference between BRANL and BRANLR is in how you specify the branch-to address: for BRANL, the address is specified in the instruction; for BRANLR, the address must be placed in the register specified by the instruction.

The BRANL Instruction: BRANL assembles into a 4-byte instruction. The branch-to address is specified in the instruction. Depending on the operands specified, the return address is placed either in a register or in the return-address stack. If the return-address stack is used, LRETURN should be used to return control to the next sequential instruction and clear the address from the stack.

The BRANLR Instruction: BRANLR assembles into a 2-byte instruction. The branch address must be stored in the branch register before BRANLR is executed. The return address is placed in the specified register (1-15) or the return-address stack.

Instructions that Return Control

You may use the IRETURN and LRETURN instructions to return control following one of the Branch-and-Link instructions.

The LRETURN Instruction: LRETURN assembles into a 2-byte instruction. When executed, it removes the latest entry from the return-address stack (last in, first out) and gives control to the instruction at that address.

Return Addresses: For both BRANL and BRANLR, the return address (the location of the next sequential instruction) is placed in a register if one of registers 1 through 15 is specified, or in a specially reserved portion of Segment 1 called the *return-address stack* if register 0 or no register number is specified. The return address is stored only if the branch is taken. When a register is specified, the return address is placed in the rightmost 2 bytes of the register.

The return-address stack is managed by the controller on a last-in first-out basis. The controller uses the SMSLSE field to keep a count of the number of entries in the stack. When a BRANL or BRANLR instruction adds an entry to the stack, the controller increases SMSLSE by 1.

When an LRETURN instruction is executed, the controller decreases SMSLSE by 1 and executes the instruction whose address was placed in the stack last.

The IRETURN Instruction: The IRETURN instruction is used in conjunction with the ADRLST and one of the branch-and-link instructions.

The Execute (LEXEC) Instruction

The execute instruction, LEXEC, is used to cause out-of-sequence execution of a single instruction (the subject instruction). The next sequential instruction after LEXEC is performed following execution of the subject instruction (except when the subject instruction is an LEXIT, LSEEK, or an instruction that modifies the instruction counter). The subject instruction may be any valid executable instruction except a jump, branch, or another LEXEC instruction.

The subject instruction is created by taking an instruction pointed to by LEXEC (the addressed instruction) and ORing it with all or part of the data in a register. The resulting subject instruction is then executed. Both the operation code and operands may be modified except for the immediate data field of the WRTI instruction. Specifying register 0 suppresses the ORing operation and causes the addressed instruction to become the subject instruction.

The subject instruction exists only for a single execution, the addressed instruction and the register are not changed by the ORing.

Storage Management Instructions

You can define segment storage in your program by using the DEFSTOR (Define Segment Storage) instruction. You can allocate segment storage by the SEGALLOC and SEGFREE, and by the DTACCESS and DTAFREE instructions.

Other 4700 Instructions

Storage Initialization Instructions

Your primary application program can initialize segment storage, that is you can set storage to some value or values, by the SINIT (Start Initialization), INITSEG (Initialize Segment), and ENDINIT (End Initialization) instructions. This initialization will take place when your program is loaded into storage.

Scratch-Pad Instruction

The scratch-pad area (SPA) is a global section of user storage that is separate from segmented storage, and accessible to all application programs. The SPA can be used to contain a dynamically-changing table, temporary data storage, or a shared area for communication between work stations.

Timer Control Instructions

The 4700 provides three timers for program control and dating, or "time-stamping" of system activities and data.

The time-of-day timer is a variable-format value representing year, month, day, hour, minute, and second. Your program may set, adjust, and read the time-of-day timer with the LTIME instruction. You can also expand the timer format to contain such values as the complete month (such as January) and

weekday (such as Tuesday). Your program can read the expanded timer format with the optional LTIMEV and LTIMET instructions. The resulting values of either format are available to your program for either dating or branching. The interval timers measure the elapsed time, or intervals, of a program execution or some other event. Intervals can be as short as 1/256 of a second with the optional INTMR instruction.

The high-resolution counter (HRC) measures program activity in intervals of 1/256 of a second, and can be used as a time-stamp or to control your program execution. Your program must define a field to contain the timer value when it is read with the LHRT instruction.

The Dump Instruction

To write any segments or files to a diskette data set use APBDUMP. The data set must be defined during the configuration process. When the dump is complete, you can use the system monitor to send the dump data set to the host.

If an APBDUMP instruction is included in the application program, the DEFDMP and COPY DEFSMS instructions must also be included. These instructions define the buffer used by APBDUMP and provide the locations of Segment 1 fields required by APBDUMP.

APBDUMP can be used in the program check routine so that a listing of the registers (Segment 0), the location of the instruction that caused the program check (Segment 1), and other desired information is available for debugging.

Note: The Extended Disk and Diskette Access Method (EDAM) is required in order for you to use the APBDUMP instruction. You will find further information in the *IBM 4700 FCS Controller Programming Library: Volume 6: Control Program Generation and in Volume 2: Disk and Diskette Programming.*

Chapter 4. Coding Rules

This chapter describes the rules by which 4700 application programs must be written. The coding rules governing the coding of 4700 assembler instructions are a limited set of the same coding and syntax rules used by the OS/VS-DOS/VS-VM/370 assembler language. The following sections of this manual explain the limitations on those assembler coding rules that you must observe when writing 4700 application programs. The prerequisite assembler publications listed in the preface of this manual describe the detailed coding and syntax rules that apply to the 4700 assembler language as well as error messages that could occur during assembly of your program.

Syntax Notation

Syntax Notation Key

We use a uniform notation to describe the syntax of the controller symbolic instructions. The notation indicates which operands you must code and which are optional, the options that are available for expressing values, the values assumed by the system if you don't code an operand, and the punctuation.

CAPITAL LETTERS
Capital letters indicate values that you must enter exactly as shown.

lowercase letters
Lowercase letters indicate where you are to insert a number, character string, or keyword in place of the lowercase letters.

punctuation .,=’()
The period, comma, equal sign, single quotation mark, and parentheses are punctuation that you must code exactly as shown. These punctuation marks separate the operands of the instructions. You need not code a comma preceding a keyword parameter for the *first* parameter in the operand field. Parentheses are sometimes optional, see "ellipsis" below.

brackets []
Brackets indicate that you can choose not to code the elements and punctuation they enclose; the operand is *optional.*

braces {}
Braces indicate that you must code the elements and punctuation they enclose; the operand is *required.*

selecting options
When you choose from more than one operand, the choices appear like this, with vertical bars separating them:

```
[1|2|3]   or   {1|2|3}
```

or they might appear stacked, like this:

```
⎡1⎤        ⎧1⎫
⎢2⎥   or   ⎨2⎬
⎣3⎦        ⎩3⎭
```

underscoring

We underscore a value to indicate that if you do not code a value for the element, the system *assumes* the underscored value. The value that the system assumes is called a *default*. In the following examples, if you do not code TYPE, the system uses *TYPE=1*.

```
[TYPE={1|2|3}]     or     [ TYPE= { 1 } ]
                                  { 2 }
                                  { 3 }
```

ellipsis...

Ellipsis points indicate that you can add one or more additional operands or sets of operands, each having the same format. For example,

```
CASE=(element1,...)
```

indicates that you can repeat the syntactical unit (element) preceding the ellipsis. The parentheses are not needed if you code only one element.

Specifying Operands

Generally, a 4700 assembler instruction performs an operation on two operands and replaces one of them with the result. Operands for these instructions are designated operand 1 and operand 2. For example, an instruction performing addition adds operand 1 and operand 2 and replaces operand 1 with the result. Operand 2 remains unchanged.

Note: When an instruction has only one principal storage reference, the suffix (1 or 2) has an additional significance. In such an instruction, the suffix identifies which part of an Index Register Number Table (IRNT) applies to the storage reference. For example, the COMP instruction has one storage reference - identified as 'operand 2'. This means that the second half of the IRNT applies to the COMP instruction. See "Segment Indexing" in Appendix F for more information.

This section of the chapter describes how these operands, which can be either defined constants or the contents of registers and/or fields, are designated in the assembler instructions.

operand 1

Is the value shown in the instruction notation having a suffix of 1. Examples are:

defcon1
defld1
(defrf1)
reg1
(reg1)
seg1
seg1,disp1
seg1,disp1,len1
immdata1

Each of these are variables that represent a way of selecting a constant or value located in a field or register that is operand 1, and is defined later in this section. When the operation creates a result, the area containing operand 1 usually contains the result unless otherwise noted in the instruction description. One or more operand 1 designators can be specified for an instruction - any one of which can be used to define the operand. They are described collectively, however, as "Operand 1" in the instruction description.

operand 2

Is described by one or more variables in the syntax notation having a suffix of 2. Examples are:

defcon2
defld2
(defrf2)
reg2
(reg2)
seg2
seg2,disp2
seg2,disp2,len2
immdata2

As in operand 1, one or more of these variables in the syntax notation of the instruction description means that operand 2 may be defined as noted; the variables are described collectively as "Operand 2". Operand 2 is not normally changed by the operation unless as described in the instruction description.

The rest of this section describes the variables that select an operand or operands. The operand suffix is disregarded in these descriptions, because most of these variables can be used to define either operand.

The operands listed below appear in several controller symbolic instructions. A subset of the OS/VS or DOS/VS assembler constants is used to code these operands. (Refer to the latest edition of *OS/VS-DOS/VSE-VM/370 Assembler Language* for detailed information

on constants.) The following descriptions include the constants that are valid for the operands, unless specified otherwise in the description of the individual instructions.

defld

This variable represents the label of a DEFLD (Define Field) instruction. DEFLD instructions represent storage areas rather than specific values themselves, but the storage area is usually initialized to a specific value. DEFLD instructions always are coded in the data definition area of the program.

defcon

This variable represents the label of a DEFCON (Define Constant) instruction. DEFCON instructions define actual values, and cannot be fields that are changed by the instruction. DEFCON instructions are always coded in the data definition area of the program.

label

This is the label of an instruction, such as a DEFLD, an executable instruction that is the target of a branch, or the name of a table. *Label* must be a character string from one to eight characters (unless specified otherwise in the descriptions of the individual instructions). The *label* operand follows the rules governing assembler labels; all characters must be alphameric, and the first character must be alphabetic. For further discussion of labels, refer to the latest edition of *OV/VS-DOS/VSE-VM/370 Assembler Language.*

reg

This variable selects one of the sixteen registers available to this application program. The value for reg can be 0 through 15, and can be specified as an unsigned decimal integer, or the label of an EQUATE instruction that refers to an unsigned decimal integer.

seg

This variable selects one of the 16 segments that the program can address. The value can be 0 through 15, as long as the corresponding segment has been defined for the work station. The value 0-15 can be an unsigned decimal integer, or the label of an EQUATE instruction that defines the integer.

disp

This variable defines the displacement into a segment. The value of disp cannot be larger than the size of the related segment. The variables seg and disp are usually used together to define a field. Specify disp as an unsigned decimal integer ranging 0 to 65 534, or as the label of an EQUATE instruction that defines the integer.

len

This variable defines the length of a field in a segment. The beginning of the field is usually located by seg,disp. The allowable maximum that len can be varies according to the instruction. Specify len as an unsigned decimal integer or the label of an EQUATE instruction that defines the integer.

(reg)

This variable defines a register containing an address that locates a storage area. Register addresses have the following format:

Bits	***Contents***
00-07	Set to Zero
08-11	Segment space ID
12-15	Segment
16-31	Length
32-47	Displacement

Register addresses can be created in a register by the LDRA (Load Register Address) instruction. You can use registers to pass data addresses from your program to another. Specify reg as an unsigned decimal integer or the label of an EQUATE instruction specifying the integer. The parentheses must be coded.

(defrf)

This is the label of a DEFRF instruction. The DEFRF instruction identifies a register (containing a register address), a displacement, and a length. The displacement to the storage area is calculated by adding the displacement in the register to the displacement from the DEFRF instruction. The length of the storage area is the length in the DEFRF. DEFRF instructions can be used to define multiple dummy section (DSECT) overlays. The parentheses must be coded.

immdata

Defines immediate data, usually as operand 2, that becomes part of, and is an operand of the instruction with which it is used.

Immdata must be 1 or 2 bytes of data specified as one of the valid types discussed below or you can specify the label of an EQUATE instruction. Immdata can also be a decimal value such as: 4 or 16. With DOS/VS there is a limit of eight characters in the immediate data including the descriptors. For example, B'10101' is counted as eight characters, because B and the single quotation marks are included in the count.

Note: If a single quotation mark (') or ampersand (&) is included in a data string, two quotation marks or ampersands must be coded in order to obtain the desired data. For example, if the data wanted is the word **can't** it must be coded as **can''t**. Similarly, if the data wanted is the phrase **one & two,** it must be coded as **one && two.**

data

Is any character (C), hexadecimal (X), binary (B), fullword (F), or halfword (H) specification in one of the following forms:

dddt'xxx...x'
tLnn'xxx...x'
tL.nn'xxx...x'

where *ddd* is a decimal number indicating the number of times the constant is to be generated (if only a single constant is required, this number is not needed); *t* is one of the valid types (C, X, B, F, or H); *nn* is a decimal number indicating the actual length of the constant; and *xxx...x* is the data

that makes up the constant, enclosed in single quotation marks for all types except address constants, which are enclosed in parentheses.

An address (A or Y) specification may also be used, but must be in the form:

dddA(label-apbname) or dddA(label-label)
ALn(label-apbname) or ALn(label-label) or ALn(label)
AL.n(label-apbname) or AL.n(label-label)
dddY(label-apbname) or dddY(label-label)
YLn(label-apbname) or YLn(label-label) or YLn(label)
YL.n(label-apbname) or YL.n(label-label)

when using the standard OS/VS and DOS/VS assembler. *label* is the label of the instruction and *apbname* is the CSECT name. If an address specification is not in this form, the assembler builds an RLD entry which will be rejected by the FORMAT service program.

Notes:

1. The above forms of data specification are the only ones valid for a controller application program. Any other forms may produce unexpected results.
2. AL3 and AL4 address constants have specific meanings for the Host Support program.

ccmask

Must be a hexadecimal value specified as X'*xx*' (the rightmost 4 bits of the hexadecimal value are used as the mask; the other 4 bits are ignored); a binary value specified as B'*nnnn*', a mnemonic (refer to the next section for a list of the mnemonics representing coded values), or the label of an EQUATE instruction that is associated with one of these values.

Labels and Mnemonics

To avoid possible conflict, the labels of the standard definitions copied by the COPY instructions should not be repeated as labels of controller symbolic instructions.

Mnemonics are used in several ways in the 4700 assembly language; as a representation of data sets and logical work stations and as a mask in a branch or JUMP instruction. A mnemonic used in a branch or JUMP instruction represents a coded value that is the value of a condition code that may have been set. The following are the mnemonics that have special meaning when coding a controller application program.

Coded values for files:

A	Absolute address
C	Composite file
CR	Control Record
DSID	
DSK	Disk or diskette
L	Log
P	Permanent file
PBN	Physical Block Number
PLR	Data set logical record
TF1	
TF2	Temporary Files
TF3	
TF4	

Coded value for a logical work station

ST	Station

Coded values for masks:

Mnemonic	Hex Value	Explanation
BL	08	A significant (one) bit is lost.
BU	04	The device is not available (busy).
EQ	01	The values compared are equal.
GE	05	The first operand is greater than or equal to the second operand.
GT	04	The first operand is greater than the second operand.
ID	02	The ID specified is invalid, the name was not found, or the ID was out of the range of valid IDs.
IL	02	An incorrect length is specified.
IO	04	An invalid Segment 0 is specified: Segment 0 operator A has been set as the default.
IS	04	An invalid segment is specified in the parameter list.
LE	03	The first operand is less than or equal to the second operand.
LT	02	The first operand is less than the second.
MD	08	A modulus error occurred.
ME	04	The tested field and mask are identical.
MO	08	All tested bits are 1's.
MX	02	The tested bits are mixed 1's and 0's.
MZ	01	All tested bits are 0 or the mask bits are all 0.
NE	06	The values compared are not equal.
NG	02	The result or data is negative.
NL	01	No significant (one) bits are lost.
NN	04	The field is not numeric.
NO	08	No Segment 0 for operator B exists.
NS	04	There is an invalid device specification.
NZ	02	The result or data is nonzero (logical instructions only).
OK	01	The operation is successful.

Mnemonic	Hex Value	Explanation
OV	08	An overflow occurred.
PS	04	The result or data is positive.
SP	04	There is insufficient segment space.
ST	02	Status is returned.
SU	08	The segment is in use.
TR	08	Truncation occurred.
ZD	08	Division by zero was attempted.
ZO	01	The result or data is zero.

The mnemonics listed above are those that are available for branching instructions. If none of these mnemonics are meaningful to the branch operation, use an EQUATE instruction to define another mnemonic.

These mnemonics may also be used as labels, but when they are used in ways other than those stated in the descriptions of the individual instructions, they may cause an error or an unexpected value to be assembled. For example, if a mnemonic is coded in an instruction for which the mnemonic is not valid, an assembler error occurs. The mnemonics are defined to represent specific coded values and have meaning only for the instructions that specifically define their use. The mnemonic representation of a coded value may be defined by some other means, such as an EQUATE, but the coded value defined by an instruction has priority over this other mnemonic definition. For example, if the following is coded:

```
EQ    EQUATE    X'50'
      BRAN      EQ,ADDR
```

no assembler error is indicated, but the EQ in the BRAN instruction will be assembled as hex 01 not hex 50.

If, however, the following is coded:

```
EQ    EQUATE    X'50'
      TSTMSKI   FLD,EQ
```

'EQ' will become hex 50 because TSTMSKI does not define a coded value for this mnemonic.

Chapter 5. 4700 Instruction Descriptions (Alphabetically)

The following are non-executable instructions. They are *not* machine instructions and do not appear within the assembled application program.

- APOPT
- BEGIN
- COPY
- DEFLD
- DEFRF
- DEFSTOR
- ENDINIT
- ENDSEG
- EQUATE
- FINISH
- INITSEG
- Ln
- LDSECT
- LEJECT
- LEND
- LSPACE
- PLPCMD
- PRINTI
- REBASE
- SAVEBASE
- SECTION
- SEGCODE
- SINIT
- USEBASE

The following are non-executable instructions. They are *not* machine instructions but they do become data within the application program and you must place them properly. For example, DEFCON instructions cannot be interspersed with executable instructions unless you branch around them.

- ADRLST
- DEFCON
- DEFDEL
- DEFDMP
- LSEEKPL
- LTIMET
- LTRTBEG
- LTRTENT
- LTRTGEN
- MASK
- MOD
- TABLE

ADDFLD--Add Field

ADDFLD algebraically adds the binary value from a field to the binary contents of a register and places the result in the register. The leftmost sign bit of the result is propagated to the left in the register.

The length of the field must not exceed 6 bytes. The leftmost bit of the field is the sign. If the length is 0, data in the register is not changed, but the condition code is set according to that register data.

Name	Operation	Operand
[label]	ADDFLD	reg1, { defcon2 / defld2 / (defrf2) / (reg2) / seg2,disp2,len2 }

```
                          ⎧ defcon2           ⎫
                          ⎪ defld2            ⎪
[label] ADDFLD    reg1,  ⎨ (defrf2)          ⎬
                          ⎪ (reg2)            ⎪
                          ⎩ seg2,disp2,len2   ⎭
```

operand 1
A register to which operand 2 will be added.

operand 2
A field to be added to operand 1. The field length is 0-6 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overlfow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, or 27 can be set.

ADDFLDL--Add Field Logical

ADDFLDL adds the contents of a 6-byte field to a register. If the field is less than 6 bytes in length, it is functionally extended to 6 bytes before addition by propagating zeros. ADDFLDL then adds the binary value from the field to the binary contents of the register and places the result in the register.

The length of the field must not exceed 6 bytes. The leftmost bit of the result is the sign. If the length is 0, data in the register is not changed, but the condition code is set according to that register data.

Name	Operation	Operand
[label]	ADDFLDL	reg1, { defcon2 / defld2 / (defrf2) / (reg2) / seg2,disp2,len2 }

operand 1
Is a register to which operand 2 will be added.

operand 2
Is a field to be added to operand 1. The field length must be 0-6 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overlfow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, or 27 can be set.

ADDREG--Add Register

ADDREG algebraically adds the binary contents of two registers and places the result in the register specified by *reg1*.

Name	Operation	Operand
[label]	ADDREG	reg1,reg2

operand 1
A register to which operand 2 will be added.

operand 2
A register containing the value to be added to the first operand.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overlfow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks: None are set.

ADDZ--Add Zoned Decimal

This instruction adds the zoned decimal value in operand 1 to the zoned decimal operand 2 value, and replaces operand 1 with the result. The length of either operand is 1-63 bytes; operands greater than 15 bytes long must be specified using register addressing.

Note: This is an optional instruction, and requires that module P31 be specified on the OPTMOD configuration macro.

Name	Operation	Operand
[label]	ADDZ	{ defld1 | (defrf1) | (reg1) | seg1,disp1,len1 } , { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Defines a zoned decimal field to which operand 2 will be added. This field cannot be in Segment 14; nor can the label of a DEFCON be specified. If the result is less than the size of operand 1, each remaining high-order byte is filled with hex F0.

operand 2
Defines a zoned decimal field to be added to operand 1.

Condition Codes: The following can be set.

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overlfow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, 09, or 27 can be set.

ADRLST--Return Address List

ADRLST creates a list of return addresses to which the IRETURN instruction will refer.

Name	Operation	Operand
[label]	ADRLST	[addr-1,addr-2,...]

addr-n
Is a label. The maximum number of labels allowed is 7. The operands are positional. If you omit one operand then the address of the next sequential instruction (NSI) will be created in that position of the address list. If you code no labels then one address will be created pointing to the next sequential instruction.

The format of the address list is as follows:

```
Byte
      ------------
 0    | FF |count|
      -----------
 2    |  addr-1  |
      ------------
           .
           .
      ------------
      |  addr-n  |
      ------------
```

AND--AND Field

You may use AND to 'AND' from 1 to 255 bytes of field 2 with the same number of bytes in field 1 and place the result in field 1. The length of field 2 determines the length of the operation.

When the AND is performed, a bit in the result is set to 1 if the corresponding bits in both fields are 1's.

Name	Operation	Operand
[label]	AND	{defld1 | (defrf1) | (reg1) | seg1,disp1} , {defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2}

operand 1
Defines a field to which the second operand will be ANDed. The segment number cannot be 14.

operand 2
Defines a field to be ANDed to the first operand. The length (1-255) determines how many bytes are in the AND operation.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is all 0's.
02	NZ	The result is mixed 1's and 0's, or all 1's.

Program Checks (hex): 01, 02, 03, or 27 can be set.

ANDI--AND Field Immediate

ANDI is used to 'AND' 1 or 2 bytes of immediate data with field 1 and place the result in field 1.

When the AND is performed, a bit in the result is set to 1 if the corresponding bits in both the field and the immediate data are 1's.

Name	Operation	Operand
[label]	ANDI	{ defld1 | (defrf1) | (reg1) | seg1,disp1 } ,immdata2

operand 1
Defines a field to which the immediate data will be ANDed. The segment number cannot be 14.

operand 2
Is 1 or 2 bytes of immediate data. The length of the immediate data determines the length of the operation.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is all 0's.
02	NZ	The result is mixed 1's and 0's, or all 1's.

Program Checks (hex): 01, 02, 03, or 27 can be set.

APBDUMP--DUMP Segment or File to Diskette

This instruction writes the contents of one or more segments, files, or the system log to a diskette data set that can be allocated during operating diskette creation. When the dump is completed, APBDUMP restores all register and SMS field values except register 15. Operation resumes with the next instruction following APBDUMP.

Notes:

To use this instruction you must code:

1. the EDAM operand on the FILES configuration macro
2. the COPY DEFSMS instruction - *before* a DEFDMP
3. the DEFDMP instruction to reserve a 454-byte area at the beginning of a segment.

Name	Operation	Operand
[label]	APBDUMP	{ { seg / (seg,...) } [,FILE] / FILE } [,ID=dumpid]

seg

Is the number of the segment to be dumped. The segment numbers are specified in ascending order.

Note: If the segment specified is the same as the buffer segment specified in DEFDMP, no dump of this segment occurs.

list

Is the list of files to be dumped. There is no dump for composite files; the list can include any combination of the following:

TF1,TF2, TF3, TF4	for temporary file subdivisions
L	for the log
P	for the permanent file

Note: At least one operand (seg or FILE) must be coded with APBDUMP.

dumpid

Is a unique identifier for this dump.

Condition Codes: This instruction may modify the condition code, however; any condition code returned will have no significance.

Program Checks (hex): 01, 02, 03, 09, 11 may be set.

APCALL--Call Assembler Application Program

This instruction calls and passes control to another 4700 assembler application program. The called program begins execution with shared or private work-station storage.

APCALL requires a 12-byte parameter list with the following format:

Bytes	Function
0-7	Called program name
8-9	Shared segment flags
10-11	Register flags

The called program name must be the same as that specified by the APBNM parameter of the called program's BEGIN statement.

The 16 bits of the shared segment flags field correspond, from left to right, to the application program's Segments 0-15. To share a segment between the calling and called application program, the corresponding flag bit must be set to 0. If a bit is set to 1 and the called application program's DEFSTOR statement defines that segment, a new segment will be allocated. Regardless of their flag bit settings, Segments 1 and 15 are always shared; Segment 13 is preallocated and shared across all stations; and Segment 14 is always allocated.

New initialized segment headers are created for newly allocated segments. For shared segments, the calling program's segment headers are passed unchanged by APCALL. The 4700 also creates segment headers for Segments 1 and 15.

The bits in the 2-byte register flag field correspond, from left to right, to the calling program's sixteen 6-byte registers. If Segment 0 was not passed (that is, it was allocated for the called program) and a register flag bit is 1, the corresponding register's contents are copied to the equivalent register for the called program. Segments and registers can be shared in any combination between the calling and called programs. For example, a register can contain a register address pointing to a data area of a dummy section (DSECT) in a common shared storage segment.

Note: Shared segments will pass data in both directions across the APCALL/APRETURN interface, but data in passed registers (Segment 0 not shared) will not be returned to the calling program.

The called program may reside in controller storage or may reside on diskette or disk until called. If programs are to be transient you must include the required macros in your system configuration. See the APLIST and TRANPL configuration macros in the *IBM 4700 Finance Communication System Programming Library: Volume 6*.

APCALL also does the following:

1. Creates a segment space ID for the called program.

2. Controls the base of the return address stack for each segment space ID.

3. Saves the alternate delimiter table pointer (SMSDEL) and the delimiter control mask (SMSDCB). SMSDEL will be initialized to zero for the called program.

4. Saves and clears any segment indexing that may be active.

Name	Operation	Operand
[label]	APCALL	{ defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2 } [,WAIT={Y|N}]

operand 2
is a field containing the 12-byte parameter list. Any length specified for operand 2 is ignored.

WAIT
controls whether or not the station will wait if storage is not available to load a non-resident program. If you specify WAIT=N and storage is not available, an immediate return is made to the calling program with a condition code of 04. The default is WAIT=Y which causes the station to be put in wait status until storage becomes available. The WAIT parameter is ignored for a resident program.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	Instruction executed successfully. This condition code is actually the result of an APRETURN instruction because if APCALL is successful, control is given to the called program. The called program performs an APRETURN and control returns to the calling program.
02		Status is stored; the status code is contained in SMSDST. See the *IBM 4700 Controller Programming Library: Volume 2* for explanation of the status codes.
04		WAIT=N is specified and there is no available area to load the requested transient application program.

Program Checks (Hex): 01, 02, 04, 09, 20, 21, 22, 23, 24, 25, 27, and 28 can be set.

Programming Notes

1. You must establish interrupt handling conventions between calling and called programs. For a given station, asynchronous interrupts that are not processed by a called program are held until another program gets control. The program that gets control must have an interrupt handling routine or an appropriate entry point defined. This program should be able to determine why the interrupt occurred. Alternatively, the original program should process the interrupt.

   If asynchronous interrupts occur while the logical work station is idle and the current application program has no entry point defined, the interrupts are held until the program is redispatched and can clear them. However, because no processing can occur, the station remains idle or appears locked up.

2. If the called application's program check routine resets the link stack counter (SMSLSE), it must not be zeroed. Its contents must be equal to the current link stack base value (SMSLSB).

3. The SMSDEL field is zeroed for the called application program and restored by APRETURN. SMSDCB is passed unchanged to the called program, but is restored by APRETURN.

4. All called application programs, whether transient or resident, may also have overlays.

5. The space allocated to a particular segment space ID may be permanent or temporary as determined by the USE parameter of the DEFSTOR instruction. If you specify USE=STATIC the area allocated, when that program is called the first time, is retained unchanged (segments, segment headers, and registers) until the controller is IPLed again.

APOPT--Application Program Options

The APOPT instruction specifies application program assembly options. APOPT must be the first assembler instruction in each assembly, and can be specified only once.

Name	Operation	Operand
[label]	APOPT	[RELOC={Y|N}] [,SPLIT={Y|N}]
		[,DIRNAME=name, {NEW|OLD}]
		[,DISP={NEW|OLD}] [,YL2={Y|N}]
		[,INDEX={nnn|0}] [,DISP16={Y|N}]
		[,REFRESH={Y|N}]

RELOC

Specifies whether the relocate option is selected allowing modules to be used in the LINKAPB function (Y) or no relocate and not allowing the modules to be used in the LINKAPB function (N).

SPLIT

Specifies whether the application program is split. If your application is to be split, see Appendix F, otherwise do not code this operand.

DIRNAME

DIRNAME=(name,NEW) specifies that the named directory should be created on the host library, and that directory entries for all sections in the assembly should be added to the new directory. If a directory with the same name already exists, another directory will not be created and the entries for the sections will not be added to the old directory.

DIRNAME=(name,OLD) specifies that the named directory already exists and that directory entries for all sections in the assembly should be added. (An entry will not be added if a duplicate entry already exists in the directory.) If the directory does not exist, a new one is created and entries for all sections in the assembly are added to it.

DISP

DISP=NEW specifies that the section or sections are new and should be added to the host library only if a section with the same name does not already exist.

DISP=OLD (or an omitted operand) specifies that a section or sections may already exist with the same name(s) as the section(s) being assembled, and that the sections being assembled should replace those that exist. If a section does not exist, the new section will be added.

YL2

YL2=Y specifies that all AL2 address constants used as operands of DEFCON instructions in the following sections should be changed to relocatable YL2 address constants. Be careful when using this operand because the DEFCON instruction cannot distinguish between expressions that are not intended to be relocated and those that are. Therefore, you should manually change the source statements if there is any question about the use of AL2 address constants in your application program.

If YL2=N is specified or the operand is omitted, AL2 address constants will not be altered.

INDEX

Provides for a unique name for each CSECT as required for OVLYSEC and SEGCODE sections that are assembled with RELOC=Y option and when the sections are separately assembled. Values may be from 0 to 998. The constant CSECT has the name specified on the macro; the instruction CSECT has a generated name (BQKIN index number) that includes the number specified by INDEX. Each instruction control section increases the value by one to create a unique name.

DISP16

Specifies whether 12- or 16-bit addresses are created for DEFCON instructions (RELOC must specify Y). If DISP16 is Y, 16-bit address fields are created, regardless of the displacement size. If N (the default) is specified, the displacement (12- or 16-bit) determines the field size.

REFRESH

REFRESH=Y means that this program is read-only; that is, it is not dynamically modified by any other program. It also means that the 4700 may reuse the main storage occupied by this program when it is not operating. The 4700 will reload the program into storage when it is to be referenced subsequently. REFRESH=N means that this program may be dynamically modified and will *not* be reloaded by the 4700.

If a program contains an OVLYSEC instruction then it will be assembled as if you specified REFRESH=N.

APRETURN--Return to Calling Program

This instruction is issued by a called 4700 assembler application program to return control to the calling program. Control is returned with the following conditions set:

1. Allocated segments belonging to this program are released unless its DEFSTOR statement specified USE=STATIC.

2. The current segment space ID is restored to reflect what had originally been assigned to the calling program by the controller.

3. The return address stack pointer, and alternate delimiter table address, and the delimiter control mask are restored.

4. Segment indexing is restored if it was active.

5. Control passes to the instruction following the APCALL instruction in the calling program.

Name	Operation	Operand
[label]	APRETURN	

Condition Codes: The condition code is always set to hex 01 (mnemonic OK).

Program Checks (hex): 26 can be set.

Programming Notes

1. If an APRETURN frees storage for which another station has been waiting, then the storage will be allocated to the other station and the request for that station will be processed.

2. You can test the condition code of APCALL in an instruction following the APCALL instruction. If the APCALL fails (for example, a diskette read error of a transient program) then the condition code will *not* be hex 01. If the APCALL is successful then control will be given to the called program and APRETURN will set condition code hex 01 when control is returned to the calling program.

BEGIN--Assembly Control

BEGIN identifies the beginning of a controller application program during assembly and builds a CSECT with the name identified by APBNM=name, an ADDMEM statement, and a control block that is used to identify the controller application program and its entry points. At least one asynchronous entry point must be specified. A 48-byte area is reserved at the beginning of each controller application program whether or not all operands are specified. BEGIN can be preceded by APOPT only.

Name	Operation	Operand
[label]	BEGIN	APBNM=(name [,{vn|1}]) ,DATE=mmddyy [,PC=label] [,DEL=label] [,STP=label] [,APENTRY=label] [,API=label] [,ATD=label] [,ACP=label] [,AST=label] [,NUMOVLY=n] [,ATM=label] [,ALA=label] [,INSNAME=name] [,DSECT={Y|N}]

APBNM

Specifies the name and version number of the controller application program:

name

Is the eight-character name of the controller application program. This name is also used in the APBNM operand of the STATION configuration macro instruction.

vn

Is the version number of this assembly (a decimal integer from 0 to 99). If it is omitted, 1 is assumed.

DATE

Is the month, day, and year of this assembly.

PC

Is the entry point to be used when the program encounters a program check. 'label' can be an external symbol when RELOC=Y is specified in the APOPT instruction.

DEL

Is the location of the delimiter table (refer to the DEFDEL instruction for a description of defining delimiters). If DEL is not specified, SMSDEL must be dynamically altered to point to the delimiter table. 'label' may be an external symbol when RELOC=Y is specified in the APOPT instruction.

STP

Is the startup entry point. After controller initialization is completed, control is passed to each logical station that uses this program and that was configured with a startup flag during the controller configuration procedure (that is, the STARTUP operand on the STATION macro instruction is specified as Y). If a startup entry point is not specified, control is not passed to a station until one of the other entry points is used. 'label' may be an external symbol when RELOC=Y is specified in the APOPT instruction.

APENTRY

Defines the label of the entry point for executing this application program when it is called by another program using APCALL.

Note: You must include a DEFSTOR instruction in your program in order to use the APENTRY operand.

API

Is the label of the entry point to be used when a program interrupt (LPOST) is presented to a station that has relinquished control by an LEXIT instruction. The 'label' may be an external symbol when RELOC=Y is specified in the APOPT instruction.

ATD

Is the entry point to be used whenever an asynchronous operation is started on a device assigned to a station using this controller application program (for example, an operator starts typing on a 4704 to initiate a transaction). The device can be at any Logical Device Address (LDA) that was specified for asynchronous input. This entry point is used only when a station has relinquished control by means of an LEXIT instruction or has never been in control. 'label' may be an external symbol when RELOC=Y is specified in the APOPT instruction.

ACP

Is the entry point to be used whenever the central processor issues an asynchronous write to a station that has relinquished control by means of an LEXIT instruction or that has never been in control. 'label' may be an external symbol when RELOC=Y is specified in the APOPT instruction.

AST

Is the entry point to be used whenever one station transmits data to another station that has relinquished control by means of an LEXIT instruction or that has never been in control. 'label' may be an external symbol when RELOC=Y is specified in the APOPT instruction.

NUMOVLY

This operand is not used. It is included for compatibility reasons only.

ATM

Is the asynchronous entry point to be used when a station's timer request is honored. A timer request is generated when the station is idle, and the SMS timer (SMSTMR) is not 0 and is equal to, or less than, the GMS timer (GMSTMR). If no other asynchronous requests are pending, the station is given control at the entry point specified by the ATM, and the SMSTMR is reset to 0. If other asynchronous requests are pending, the timer request is canceled, but the SMSTMR value is unchanged. If timer entry processing is not desired, the SMSTMR value should remain at 0. A program check may result in SMSTMR is set to a nonzero value when ATM is not specified. 'label' may be an external symbol when RELOC=Y is specified in the APOPT instruction.

ALA

Is the Alternate Line Attachment program entry point where 'label' is a 1-to-8-character name. Processing begins at this point when an SNA-Primary device presents data or status to an idle station. You must specify this entry point name if asynchronous entry can occur. An SNA-Primary/ALA LREAD is normally coded at this entry.

INSNAME

This operand is used only in split application programs. See Appendix F for further information.

DSECT

Specifies whether DEFLD instructions expand as DSs or EQUs. (DS and EQU are System/370 instructions.) If DSECT=Y is specified, DEFLD instructions expand as DSs within a dsect. For each segment (0 through 15) there is a unique dsect named BQK$S*x*, where *x* equals one character, 0 through F. In this case, the value field of the cross-reference listing contains the displacement of the field defined by the DEFLD instruction. If DSECT=N is specified, DEFLD instructions expand as EQUs and no dsects are formed. In the latter case, the value field of the cross-reference listing is meaningless.

Note: The label of an EQUATE instruction cannot be used in the operands for the BEGIN instruction.

Programming Notes: Asynchronous entry point priorities are as follows:

1. CPU message pending
2. ALA message pending
3. terminal message pending
4. station message pending
5. program interrupt pending
6. timer interrupt pending.

BRAN--Branch

BRAN conditionally or unconditionally changes the sequence of program execution. If any bit is set to 1 in the mask that is specified in the instruction and the corresponding bit is set in the present condition code, then the condition is satisfied, and the branch is taken. When an unconditional branch or a branch with the condition satisfied is executed, operation continues with the instruction referred to by the BRAN instruction. Otherwise, operation continues with the next sequential instruction.

Name	Operation	Operand
[label]	BRAN	[ccmask|X'F',] branch address

ccmask

Is the condition to be met for the branch to be taken (refer to the condition codes set by individual instructions). The *ccmask* can be in the form of a mnemonic (see Chapter 4 for a list of the mnemonics representing coded values), a 1-byte hexadecimal expression, a 4-bit binary expression, or the label of an EQUATE instruction expressing one of the preceding numeric values. If the operand is omitted or if X'F' is coded then the branch is always performed. If *ccmask* is specified as hex 0, the branch is never taken.

branch address

Is the label of the instruction to be executed if the branch is taken. The label may be an external symbol when RELOC=Y is specified in the APOPT instruction.

Condition Codes: The code is not changed.

Program Checks (hex): 0B can be set.

BRANL--Branch and Link

BRANL conditionally or unconditionally changes the sequence of program execution and stores the location of the next sequential instruction. If any bit in the mask specified in the instruction is set and the corresponding bit in the present condition code is set, then the condition is satisfied and the branch is taken. When an unconditional branch or a branch with the condition satisfied is executed, the location of the next sequential instruction is:

- placed in the return-address stack in the segment 1 machine section (SMS), if no register, or register 0 is specified;

  or

- placed in the register specified by the BRANL instruction;

and execution continues with the instruction to which the BRANL instruction refers. Otherwise, execution continues with the next sequential instruction.

Use the BRANR instruction to return to the next sequential instruction from a BRANL when a register is specified; and the LRETURN instruction to return otherwise. The number of entries that the station's return-address stack can hold may be specified as 0 to 255 (default=6) by coding the RETSTK operand of the STATION macro. Each BRANL, BRANLR, and LSEEKP instruction that uses the return-address stack adds one entry, and each LRETURN instruction removes one entry. It is possible to overflow the stack, which results in program check.

Name	Operation	Operand
[label]	BRANL	[ccmask|X'F',] branch address [,reg]

ccmask
Is the condition to be met for the branch to be taken (refer to the condition codes set by individual instructions). If the operand is omitted or if X'F' is coded then the branch is always performed. If *ccmask* is specified as hex 0, the branch is never taken.

branch address
Is the label of the instruction to be executed if the branch is taken. The label may be an external symbol when RELOC=Y is specified in the APOPT instruction.

reg
Is a register (1-15) in which the location of the next sequential instruction is to be stored if the branch is taken. The location of the NSI is stored in the rightmost 2 bytes of the register. The leftmost 4 bytes are set to zeros. If this operand is omitted or specified as 0, the location is stored in the return-address stack in segment 1.

Condition Codes: The code is not changed.

Program Checks (hex): 04 or 0B can be set.

BRANLR--Branch and Link Register

BRANLR conditionally or unconditionally changes the sequence of program execution. If any bit in the mask specified in the instruction and the corresponding bit in the present condition code is set, the condition is satisfied, and the branch is taken. When an unconditional branch or a branch with the condition satisfied is executed, the location of the next sequential instruction is:

- placed in the register (*reg2*) specified in the BRANLR instruction
- or placed in the return-address stack of segment 1.

If the return register (*reg2*) is not specified or is specified as 0, the branch is taken to the instruction whose location is specified by *reg1*. Otherwise, execution continues with the next sequential instruction.

When the NSI location is placed in the return-address stack, the program should issue an LRETURN instruction to return control to the NSI. When the NSI location is stored in a register, the program should issue a BRANR instruction specifying the return register to return control to the next sequential instruction.

The number of entries that the station's return-address stack can hold may be specified as 0 to 255 (default=6) by coding the RETSTK operand of the STATION macro. Each BRANL, BRANLR, or LSEEKP instruction that uses the return-address stack adds one entry, and each LRETURN instruction removes one entry. It is possible to overflow the stack. The overflow results in program check 04.

Name	Operation	Operand
[label]	BRANLR	[ccmask|X'F',] reg1 [,reg]

ccmask

Is the condition to be met for the branch to be taken (refer to the condition codes set by individual instructions). If the operand is omitted or if X'F' is coded then the branch is always performed. If *ccmask* is specified as hex 0, the branch is never taken.

reg1

Is a register (0-15) that contains the location of the instruction to be executed if the branch is taken. The location must be in the rightmost 2 bytes.

reg2

Is a register (1-15) in which the location of the next sequential instruction is to be stored. The location of the NSI is stored in the rightmost 2 bytes, and the leftmost 4 bytes of the register are zeroed. If *reg2* is not specified or is specified as 0, the return location is placed in the return-address stack in segment 1. The same register may be used for both the branch-to and the return locations.

Condition Codes: The code is not changed.

Program Checks (hex): 04 or 0B can be set.

BRANR--Branch Register

BRANR conditionally or unconditionally changes the sequence of program execution. If any bit in the mask specified in the instruction and the corresponding bit in the present condition code is set, the condition is satisfied, and the branch is taken. When an unconditional branch or a branch with the condition satisfied is performed, execution continues with the instruction referred to by the register in the BRANR instruction. Otherwise, execution continues with the next sequential instruction.

Name	Operation	Operand
[label]	BRANR	[ccmask|X'F',] reg

ccmask
: Is the condition to be met for the branch to be taken (refer to the condition codes set by individual instructions). If the operand is omitted or if X'F' is coded then the branch is always performed. If the ccmask is specified as hex 0, the branch is never taken.

reg
: Is a register (0-15) that contains the location of the instruction to be executed if the branch is taken.

Condition Codes: The code is not changed.

Program Checks (hex): 0B may be set.

BRANX--Branch on Index

The BRANX instruction specifies a register and a branch address. The register contains three 2-byte fields:

Bytes	*Field*	*Range of Values*
0,1	Comparand	0 to 65 535
2,3	Increment	-32768 to 32767
4,5	Index	0 to 65 535

The increment is added to the index and the sum is placed into the index field. The index field is compared with the comparand field. If the comparison is not equal, a branch is made to the branch address; on equal comparison, no branching occurs.

An increment value of zero defaults to minus one.

An overflow during addition is ignored and does not affect the comparison.

Name	Operation	Operand
[label]	BRANX	reg,branch address

reg
Is a register (0-15) that contains the comparand, increment, and index.

branch address
Is the label of the instruction to be executed if the updated index field is not equal to the comparand field. The label may be an external symbol when RELOC=Y is specified in the APOPT instruction.

Condition Code: The code is not changed.

Program Checks (hex): 0B can be set.

Programming Notes: The following example shows a use of the BRANX instruction for loop control. Assume a loop is to be executed three times with the index having successive values of 20, 25, and 30.

Note: Be sure that the sum of the initial index value plus the repeated additions of the increment will eventually equal the comparand.

```
CONST   DEFCON  H'35',H'5',H'20'            1
          •
        MVFXD   REG2,CONST                  2
LOOP      •
          •                                 3
          •
        BRANX   R02,LOOP                    4
```

1 Defines constants for comparand, increment, and index values.

2 Initializes register 2 for loop control.

3 Loop to be executed.

4 Performs loop function until the index in register 2 equals the comparand.

CAFLD--Compare Arithmetic Field

CAFLD algebraically compares the value of the data in a register with the value of the data in a field and sets the condition code to indicate the result. Six bytes are compared. If the field is less than 6 bytes long, CAFLD compares the field as if its length were increased to 6 bytes by the propagation of the field's leftmost bit.

Name	Operation	Operand
[label]	CAFLD	reg1, { defcon2 / defld2 / (defrf2) / (reg2) / seg2,disp2,len2 }

operand 1
Is a register containing the first comparand.

operand 2
Is a field containing the other comparand. The length of the field is from 0 to 6 bytes. If 0 is specified, the register contents are compared to 0.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The values are equal.
02	LT	The first operand is less than the second operand.
03	LE	The first operand is less than or equal to the second operand.
04	GT	The first operand is greater than the second operand.
05	GE	The first operand is greater than or equal to the second operand.
06	NE	The first operand and the second operand are not equal.

Program Checks (hex): 01, 02, 03, or 27 can be set.

CAFLDL--Compare Arithmetic Field Logical

CAFLDL compares the contents of a register with the contents of a 6-byte field. If the field is less than 6 bytes, CAFLDL compares the field as if its length is increased to 6 bytes by propagating zeros. CAFLDL sets a condition code to indicate the result of the comparison.

Name	Operation	Operand
[label]	CAFLDL	reg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Is a register containing the first comparand.

operand 2
Is a field containing the other comparand. The length of the field is from 0 to 6 bytes. If 0 is specified, the register contents are compared to 0.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The values are equal.
02	LT	The first operand is less than the second operand.
03	LE	The first operand is less than or equal to the second operand.
04	GT	The first operand is greater than the second operand.
05	GE	The first operand is greater than or equal to the second operand.
06	NE	The first operand and the second operand are not equal.

Program Checks (hex): 01, 02, 03, or 27 can be set.

CAREG--Compare Arithmetic Register

CAREG algebraically compares the arithmetic value of the contents of two registers and sets the condition code to indicate the result.

Name	Operation	Operand
[label]	CAREG	reg1,reg2

operand 1
Is a register containing the first comparand.

operand 2
Is a register containing the second comparand.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The values are equal.
02	LT	The first operand is less than the second operand.
03	LE	The first operand is less than or equal to the second operand.
04	GT	The first operand is greater than the second operand.
05	GE	The first operand is greater than or equal to the second operand.
06	NE	The first operand and the second operand are not equal.

Program Checks: None are set.

CCDI--Compare Character Data Immediate

The CCDI instruction compares the logical value of a field with the immediate operand and sets the condition code to indicate the result. Either 1 or 2 bytes are compared.

Name	Operation	Operand
[label]	CCDI	{ defcon1 | defld1 | (defrf1) | (reg1) | seg1,disp1 } ,immdata2

operand 1
Is a field to compare with operand 2.

operand 2
Is 1 or 2 bytes of immediate data. If only 1 byte of immediate data is specified, CCDI performs a 1-byte comparison. If 2 bytes of immediate data are specified, CCDI performs a 2-byte comparison.

Condition Code: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The values are equal.
02	LT	The first operand is less than the second operand.
03	LE	The first operand is less than or equal to the second operand.
04	GT	The first operand is greater than the second operand.
05	GE	The first operand is greater than or equal to the second operand.
06	NE	The first operand and the second operand are not equal.

Program Checks (hex): 01, 02, 03, or 27 can be set.

CCFLD--Compare Character Field

CCFLD compares the logical value of a segment-header addressed field with the logical value of another field, and sets the condition code to indicate the result. The length of the two fields is assumed to be the same.

When performing logical comparisons, fields are compared from left to right and the comparison ends when a difference is found or the fields are determined to be equal. The comparison is based on EBCDIC codes of the bytes having the same relative positions.

Name	Operation	Operand
[label]	CCFLD	seg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
: Is a field in the specified segment to compare with operand 2. The field location is determined by the primary field pointer.

operand 2
: Is a field to compare with operand 1. The length of the field is 0 to 15 unless you specify register addressing, which allows a length ranging 0 to 65 535. If 0, the length of the comparison is determined by the field length indicator of seg1.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The values are equal or the length of the field in seg1 is 0 and no operation (NOP) occurs.
02	LT	The first operand is less than the second operand.
03	LE	The first operand is less than or equal to the second operand.
04	GT	The first operand is greater than the second operand.
05	GE	The first operand is greater than or equal to the second operand.
06	NE	The first operand and the second operand are not equal.

Program Checks (hex): 01, 02, or 27 can be set.

CCFXD--Compare Character Fixed

CCFXD compares the logical values of two fixed fields and sets the condition code to indicate the result. The length of the two fields is assumed to be the same.

Name	Operation	Operand
[label]	CCFXD	{ defcon1 | defld1 | defrf1) | (reg1) | seg1,disp1 } , { defcon2 | defld2 | (defrf2, | (reg2) | seg2,disp2,len2 }

operand 1
Is a field to be compared with operand 2.

operand 2
Is a field to be compared with the field defined in operand 1. The length of this field is from 0 to 65 535; operands greater than 255 bytes long must be selected using register addressing. The number of bytes to compare is determined by this field length.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The values are equal or the length of field 2 is 0, and no operation (NOP) occurs.
02	LT	The first operand is less than the second operand.
03	LE	The first operand is less than or equal to the second operand.
04	GT	The first operand is greater than the second operand.
05	GE	The first operand is greater than or equal to the second operand.
06	NE	The first operand and the second operand are not equal.

Program Checks (hex): 01, 02, or 27 can be set.

Programming Notes: For example, when the new balance is received from the central processor, the application program may compare account numbers to ensure that the correct data has been received. Below are the instructions that perform the comparison.

```
CPACCT     DEFLD      CPINSEG,0,7          1
OUTACCT    DEFLD      OUTSEG,2,7           2
             •
             •
             •
CHKRESP    CCFXD      OUTACCT,CPACCT       3
           BRAN       NE,DIFRESP           4
```

1 Defines the account number within the field transmitted from the central processor.

2 Defines the account number within the field sent to the central processor.

3 Compares the two account numbers.

4 Branches to a routine that processes the message if the account numbers are not equal.

CCSEG--Compare Character Segment

CCSEG compares the logical value of two segment-header addressed fields and sets the condition code to indicate the result. The comparison starts at the primary field pointer (PFP) in each of the segments. The length of the comparison is determined by the field length indicator of the second segment. If *seg1* and *seg2* are the same segment, the comparison field of *seg1* starts at the secondary field pointer; the secondary field pointer, however, does not change.

Name	Operation	Operand
[label]	CCSEG	seg1,seg2

operand 1
Is a field in the specified segment to compare with operand 2.

operand 2
Is a field in the specified segment to compare with operand 1. The length of this field can be 0 to 65 535.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The values are equal or the length of the field in operand 2 is 0, and no operation (NOP) occurs.
02	LT	The first operand is less than the second operand.
03	LE	The first operand is less than or equal to the second operand.
04	GT	The first operand is greater than the second operand.
05	GE	The first operand is greater than or equal to the second operand.
06	NE	The first operand and the second operand are not equal.

Program Checks (hex): 01 or 02 can be set.

COBLCALL--Call a COBOL Application Program

This instruction must be used by a 4700 assembler program to call another application program written in the COBOL programming language. This instruction creates the parameter list protocol expected by the COBOL program. COBLCALL sets register 1 to the parameter list, then does an APCALL. Register 2 is used as a work register. Refer to the *4700 COBOL Programmer's Guide* for instructions and other requirements.

Name	Operation	Operand
[label]	COBLCALL	ap, { defld1 | (defrf1) | (reg1) } , { defld2 | (defrf2) | (reg2) } [, { defld3 | (defrf3) | (reg3) } ,...]

ap

Names the COBOL program being called.

operand 1

Defines the location of a parameter list containing six-byte register addresses, one for each parameter that is to be passed to the COBOL program.

operand 2, 3, and so on.

Define the parameters to be passed to the called COBOL program by COLBCALL. COBLCALL stores a register address for each parameter in the parameter list defined by operand 1.

Condition Codes: This instruction may modify the condition code, however, any condition code returned will have no significance.

Program Checks(hex): 01, 02, 03, 09, 11, and 20 - 29.

COMP--Compress and Compact

COMP compresses and compacts an input data stream, based on the information in a parameter list, and stores the results in a specified segment. See Chapter 3 for a detailed discussion of compression and compaction.

Note: COMP requires the optional P27 module, which may be included via the P27 operand on the OPTMOD configuration macro.

COMP requires information coded in a parameter list (see COPY DEFCPL) containing the following fields:

CPLINS
2-byte field containing the segment number of the input area. The input area contains the data to be processed.

CPLIND
2-byte field containing the displacement into the segment to the input area.

CPLINL
2-byte field containing the length of the input area.

CPLOUS
2-byte field containing the segment number of the output area. At the completion of COMP, the output area contains the compressed/compacted data. The segment may not be 14.

CPLOUD
2-byte field containing the displacement into the segment to the output area.

CPLOUL
2-byte field containing the length of the output area.

CPLPRI
1-byte field specifying the prime compression character (the character that will be represented by SCB type 10xxxxxx). During compression, any occurrence of from 3 to 63 repetitions of the prime character is replaced by one byte containing the compression code (10xxxxxx), where 'xxxxxx' is the count of prime character repetitions.

CPLFLG
1-byte input flag field

Bit	*Meaning*	
0	=1	Compaction requested. CPLTBS and CPLTBD must be initialized to the segment and displacement of a compaction table (see the COMPTB instruction).
0	=0	Compaction not requested.
1	=1	Compression requested.
1	=0	Compression not requested.
2-7	=0	Reserved.

Note: Compaction, compression or both functions may be requested. At least one of the functions must be requested.

CPLTBS
2-byte field containing the segment number of the compaction table information area. See the COMPTB instruction.

CPLTBD
2-byte field containing the displacement into the segment to the compaction table information area. The assumed length of the area is 257 bytes.

During operation of the COMP instruction, the controller sets the following fields in the parameter list:

CPLIND
Contains the displacement to the next input byte. If the input area is exhausted, CPLIND will point to the byte immediately following the input area.

CPLINL
Contains the remaining length of the input area. When the input area is exhausted, CPLINL contains zero.

CPLOUD
Contains the displacement to the next output byte. If the output area is full, CPLOUD will point to the byte immediately following the output area.

CPLOUL
Contains the remaining length of the output area. When the output area is full, CPLOUL contains zero.

CPLTST
Contains the completion status for COMP. When the condition code is 01, CPLTST will be zero. When the condition code is 02, CPLTST will contain a code that indicates the reason for termination of the COMP instruction.

CPLTOV (X'80'): Indicates the output for the next input byte would extend beyond the output area.

CPLTIL (X'40'): Indicates that the length of the input area was initialized to zero.

Name	Operation	Operand
[label]	COMP	{ defld2 defrf2) (reg2) seg2,disp2 }

operand 2

Is a field containing the parameter list (see the COPY DEFCPL instruction in Appendix B). The length specified is ignored because the parameter list is defined as a fixed length area. The parameter list must not be in Segment 14.

When using register addressing to locate a parameter list, the parameter list can be located in a noncurrent segment space. However, if the parameter list contains the address (segment, displacement) of other storage areas (that is, input and output areas, tables) the other storage areas are always in the current segment space.

Note: The prime compression characters for a data stream compressed at the host or some other point on the network must be passed to the receiving program using standard SNA protocols.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	The instruction was executed successfully.
02	ST	COMP was terminated. See CPLTST in the DEFCPL parameter list.

Program Checks: 01, 02, 09, or 27 can be set.

COMPTB--Build Compaction Table

COMPTB dynamically builds a 257-byte compaction table for use by the COMP instruction. See Chapter 3 and the COMP instruction for a discussion of compaction.

The COMPTB operands specify the location of the input data to be formatted into the compaction table, and the area in which the compaction table is to be built. You select the characters that will appear in the data to be compacted (the compaction set), and the subset of those characters that will appear most frequently in pairs in the data stream (the master characters). COMPTB formats this information into the compaction table for use by COMP.

Note: This instruction requires the P27 module, which may be included via the P27 operand on the OPTMOD configuration macro.

Coding Input for COMPTB: The COMPTB instruction builds a compaction table, in the correct format, from information you supply in an input area. The input area contains:

- The number of master characters
- The master characters, themselves
- The remainder of the compaction set, arranged beginning with those characters least likely to occur in the data stream to be compacted.

To begin, calculate the number of characters that can appear in the data stream. Assume a compaction set of 87 possible characters in the data stream. The following table:

Compaction Set Size	*Master Characters*
255	1
252	2
247	3
240	4
231	5
220	6
207	7
192	8
175	9
156	10
135	11
112	12
87	13
60	14
31	15
16	16

indicates that 13 master characters are used with an 87-character compaction set. So, the first byte of the COMPTB input area would be coded as:

```
MCHLGTH DEFCON X'0D'   13 master characters
```

The compaction set *must* contain a number of characters that matches exactly one of the sizes in the left-hand column (Compaction Set Size). If it does not match, the compaction set must be expanded to contain the next higher increment.

For master characters, assume that the following 13 characters will occur in the data stream most regularly in pairs:

a d e g i l n o r s t u 'space'

The next portion of the COMPTB input area would be coded as:

```
MCHARS  DEFCON  X'8184858789939596 99A2A3A440'
```

the hexadecimal equivalents of the master characters. The remainder of the COMPTB input area will be the hexadecimal equivalents of the remaining nonmaster characters in the compaction set:

```
COMPSET DEFCON  X'4C6E7C8283868891929497 98A5
                 A6A7A8A9C1C2C3C4C5C6C7C8C9
                 D1D2D3D4D5D6D7D8D9E2E3E4E5
                 E6E7E8E9F0F1F2F3F4F5F6F7F8
                 F94A4B4D4E505A5B5C5D5E6061
                 6B6C6D6F7A7B7D7E7F'
```

You may then code the COMPTB instruction referring to the input area and the output area where the compaction table will be created.

The location of the input area is indicated by the primary field pointer of the segment specified by operand 1. A length of 17 is assumed when the number of master characters (M) is 16. A length of 257 minus M x M is assumed when M is less than 16.

Name	Operation	Operand
[label]	COMPTB	seg1, { defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 1
Is a field containing input data.

operand 2
Is a field that will contain the compaction table. The field must not be in Segment 14. A length of 257 is assumed.

Condition Codes: One of the following may be set:

Hex Code	Possible Mnemonic	Explanation
01	OK	Successful execution.
04		The number of master characters chosen is zero, or greater than 16.
08		The same character occurs more than once in the compaction set.

Program Checks (hex): 01, 02, 09, or 27 can be set.

COMPZ--Compare Zoned Decimal

This instruction compares the zoned decimal data in operands 1 and 2 algebraically; neither operand is altered by the operation. The length of either operand is from 1 to 63 bytes; operands longer than 15 bytes must be addressed using register addressing. A shorter operand is padded on the left with decimal zeros (hex FO) to make it the same length as the longer operand.

Note: This is an optional instruction, and requires that optional module P31 be specified on the OPTMOD macro.

Name	Operation	Operand
[label]	COMPZ	{ defcon1 | defld1 | (defrf1) | (reg1) | seg1,disp1,len1 } , { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Is a field containing the first zoned decimal comparand.

operand 2
Is a field containing the second zoned decimal comparand.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The values are equal.
02	LT	The first operand is less than the second operand.
03	LE	The first operand is less than or equal to the second operand.
04	GT	The first operand is greater than the second operand.
05	GE	The first operand is greater than or equal to the second operand.
06	NE	The first operand and the second operand are not equal.

Program Checks (hex): 01, 02, 09, or 27 can be set.

COPY--Copy Source Code

The COPY instruction copies predefined source code into your application program during the assembly process. It can be used to make system definitions for: segments 0, 1, 14, and 15; parameter lists; and other source code - part of your program. System definitions provided by the 4700 are in Appendix B.

You can also define your own copy files and include them in the subsystem library of the 4700 Host Support. (See *4700 Host Support User's Guide*, SC31-0020.

Because system definitions are subject to change, you should refer to system information and parameter lists using the labels provided. Fixed fields that are defined within system copy files should be referred to individually, for example, COPY DEFAPB contains:

```
APBLTH  DEFLD   14,,2
APBROD  DEFLD   14,,2
```

Because the DEFAPB definition can change, you should not code the following DEFLD and expect it to contain both APBLTH and APBROD:

```
BOTH    DEFLD   14,APBLTH,4
```

COPY instructions should be included in the data definition section of your application program.

The fields defined in DEFGMS and DEFSMS are primarily intended to be read-only by your application program, however, application programs do have access to these fields. Be careful in modifying any of these fields. Modification by your application program can affect its own subsequent operation.

For a more detailed description of the fields defined by system copy files, refer to Appendix B.

Name	Operation	Operand
	COPY	copyfilename

copyfilename
Is the name of the file to be copied.

CRETN--Conditional Return (COBOL)

This instruction transfers control to another location in the program depending on the content of a two-byte location in storage. If the location contains zero, execution resumes at the next instruction following CRETN. If the location is nonzero, the value is used as a displacement from the beginning of the program to find the point where execution begins; the nonzero location is set to zero by CRETN.

This instruction has been implemented to facilitate support for COBOL and is not considered useful for general purpose for COBOL and is not considered useful for general purpose application programming.

Note: This instruction requires that module P34 be specified on the OPTMOD configuration macro.

Name	Operation	Operand
[label]	CRETN	{ defld2 / (defrf2) / (reg2) / seg2,disp2 }

operand 2
Is a field containing the two-byte value. The field must not be in Segment 14.

Condition Code: The condition code is unchanged.

Program Checks (hex): 01, 02, 09, 0B, or 27 can be set.

DECOMP--Decompress and Decompact

DECOMP decompresses and optionally decompacts a data stream based on the information contained in the DEFDCP parameter list defined by the COPY DEFDCP instruction.

Note: DECOMP requires the P26 optional module, which you may include on the OPTMOD configuration instruction.

See Chapter 3 for a discussion of decompression and decompaction.

Before issuing the DECOMP instruction to decompress/decompact a data stream, complete these fields in the parameter list:

DCPINS
Is a 2-byte field containing the segment number of the segment containing the input data to be processed.

DCPIND
Is a 2-byte field containing the displacement of the input data into the input segment.

DCPINL
Is a 2-byte field containing the length of the input data.

DCPOUS
Is a 2-byte field containing the segment number of the segment that is to contain output data. Do not use Segment 14.

DCPOUD
Is a 2-byte field containing the displacement into the output segment.

DCPOUL
Is a 2-byte field containing the length of the output area.

DCPPRI
Is a 1-byte field containing the prime compression character used during compression. Remember, the String Control Character for compressed prime characters contains only the count of repeated characters, not the character itself. This prime character must be supplied to the decompression procedure.

DCPFLG
Is a one-byte compaction request flag:

1xxxxxxx Input contains compacted data. DCPTBS and DCPTBD must also be set if this value is specified.

DCPTBS
Is a 2-byte field containing the segment number of the segment containing the decompaction table (see the DECOMPTB instruction).

DCPTBD

Is a 2-byte field containing the displacement of the decompaction table.

When DECOMP is complete, the following fields in the DEFDCP parameter list are completed to indicate the status of the operation.

DCPIND

Contains the displacement of the next SCB in the input data stream. When the data stream contains no further SCBs, this field contains the displacement of the byte immediately following the input area.

DCPINL

Contains the number of characters remaining in the input area. When the entire input stream is processed, this field contains zero.

DCPOUD

Contains the displacement of the next available byte in the output area. When the output area is full, this field contains the displacement of the byte immediately following the output area.

DCPOUL

Contains the number of bytes remaining in the output area. When the output area is exhausted, this field contains zero.

DCPTST

Contains the completion status of the DECOMP operation. If the condition code is hex 01, this field contains zero. When the condition code is hex 02, this field indicates the reason for premature ending of DECOMP.

DCPTOV (X'80')

Indicates the output area is not large enough to contain the decompressed/decompacted data for the current SCB.

When output overflow occurs, decompression/decompaction ends as if the SCB causing the overflow had not been processed. Although partially processed data from the current SCB may appear in the output area, the input and output areas' displacements and lengths (DCPIND, DCPINL, DCPOUD, DCPOUL) reflect the completion status of the last fully processed SCB.

DCPTIV (X'40')

Indicates the remaining length of the input area is not of sufficient size to contain the current SCB's associated data. For a non-compressed/non-compacted SCB or for a compact code SCB, DCPINL is not large enough to contain the SCB plus the number of bytes indicated by the SCB count field. For a repeat-next-character SCB, the remaining length of the input area is 1 (DCPINL=1) and is therefore not large enough to contain both the SCB and the byte to be duplicated.

When input overflow occurs, decompression/decompaction is terminated as if the SCB causing the overflow had not been processed.

DCPTCE (X'20')

Indicates the input area contains a compact code SCB. The parameter list does not specify input data in compact code.

Decompression/decompaction ends as if the compact code SCB had not been processed.

DCPTSL(X'10')

Indicates the current SCB count field equals B'000000'.

Decompression/decompaction ends as if the invalid SCB had not been processed.

DCPTIL (X'08')

Indicates that the length of the input area was initialized to zero.

Name	Operation	Operand
[label]	DECOMP	{ defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 2

Is a field containing the parameter list (see the COPY DEFDCP instruction). The length specified is ignored because the parameter list is defined as a fixed length area. The parameter list must not be in Segment 14.

When using register addressing to locate a parameter list, the parameter list can be located in a noncurrent segment space. However, if the parameter list contains the address (segment, displacement) of other storage areas (that is, input/output areas, tables), the other storage areas are always in the current segment space.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	Successful completion.
02	ST	Processing terminated; see DCPTST for status code.

Program Checks (hex): 01, 02, 09, 11, or 27 may be set.

DECOMPTB--Build a Decompaction Table

DECOMPTB builds a 256-byte decompaction table for use by the DECOMP instruction in decompacting a compacted data stream.

Note: DECOMPTB requires the P26 module, which may be included via the P26 operand on the OPTMOD configuration macro.

The DECOMPTB operands specify the location of the input data to be formatted into the decompaction table, and the area in which the decompaction table is to be built. The input for the decompaction table must be the same as that used to build the compaction table (see COMPTB) with which the data stream was compacted.

The input field's location is indicated by the primary field pointer. A field length of 17 is assumed when the number of master characters is 16. When the number of master characters is less than 16 the field length is assumed to be 257 less the square of the number of master characters.

Name	Operation	Operand

```
                                  { defld2       }
                                  { (defrf2)     }
[label] DECOMPTB seg1,            { (reg2)       }
                                  { seg2,disp2   }
```

operand 1
Is a field in the specified segment that contains the input used to create the decompaction table.

operand 2
Is a field into which the decompaction table will be built. The field must not be in Segment 14. DECOMPTB assumes a length of 257 bytes for this operand.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	Successful completion.
04		The number of master characters is zero or greater than 16.

Program Checks (hex): 01, 02, 09, and 27 can be set.

DEFCON--Define Constant

DEFCON becomes a data string within Segment 14 (the controller application program). It must be specified before any of the other instructions that refer to the data string. The specification of data is the same as for an assembler language DC instruction, but fewer types of data may be specified.

Note: The maximum number of DEFCONs that can be included in an assembly is 4095.

Name	Operation	Operand
[label]	DEFCON	data1[,data2]...[,datan] [,0]

data-data*n*

Is any character (C), hexadecimal (X), binary (B), fullword (F), or halfword (H) specification in one of the following forms:

dddt'xxx...x'
tLnn'xxx...x'
tL.nn'xxx...x'

where *ddd* is a decimal number indicating the number of times the constant is to be generated (if only a single constant is required, this number is not needed); *t* is one of the valid types listed above; *nn* is a decimal number indicating the actual length of the constant; and *xxx...x* is the data that makes up the constant, enclosed in single quotation marks for all types except address constants which are enclosed in parentheses.

An address (A or Y) specification may also be used, but must be in the form:

dddA(label-apbname) or dddA(label-label)
ALn(label-apbname) or ALn(label-label)
AL.n(label-apbname) or AL.n(label-label)
dddY(label-apbname) or dddY(label-label)
YLn(label-apbname) or YLn(label-label) or YLn(label)
YL.n(label-apbname) or YL.n(label-label)

When using the standard OS/VS and DOS/VS assembler, *label* is the label of the instruction and *apbname* is the CSECT name. If not in this form, the assembler builds an RLD entry. 'label' in the form YLn (label) may be an external symbol when RELOC=Y is specified in the APOPT instruction. Note that AL3 and AL4 address constants have specific meanings for the host support and should not be used to create an RLD entry. The only valid relocatable address constant is YL2 or Y(label). All other A or Y specifications should be for non-relocatable expressions.

Note: The above forms of data specification are the only ones valid for a controller application program. Any other forms may produce unexpected results.

0

Associates a length of 0 with this constant.

Programming Notes: For example, the following instructions define a message in Segment 14 and move it to an output area in Segment 2:

```
OPMSG     DEFCON    C'STOP PAYMENT'

FIELD2    DEFLD     2,10,20

            •

          MVFXD     FIELD2,OPMSG  (The number of characters moved
                                  is determined by the length of the
                                  source field, OPMSG.)
```

Note that, if DSECT=Y is specified on the BEGIN instruction, a DSECT name of BQK$Sn (where n is the segment number from 0 to F) will be generated for each segment referenced in a DEFLD or DEFCON instruction. This DSECT name can then be used to generate values with absolute expressions. For example, if the following instructions are coded:

```
INPUT DEFLD    4,2,8
      DEFCON AL2(INPUT-BQK$S4)
```

the DEFCON will generate a two-byte address of the beginning of the field name INPUT into Segment 4.

DEFDEL--Define Delimiters

DEFDEL becomes a list of characters that will be recognized as field or message delimiters. The use of these characters is controlled by an associated mask and the delimiter control byte in Segment 1 (SMSDCB field). If any corresponding bits (between the delimiter control byte and the mask in the instruction) are both set (=1), the delimiter is recognized as valid. The set of delimiters used can be altered by changing the delimiter control byte.

Only one DEFDEL instruction is used in a controller application program when DEL is specified in the BEGIN instruction. If DEL is not specified, multiple DEFDELs may be used. The address of the delimiter table must be specified by:

- setting SMSDEL with the displacement into Segment 14 of the beginning of a delimiter table **or**
- loading the address of a delimiter table into a register, setting the register number in SMSDRG, and setting SMSDEL to hex FFFF.

The station saves and restores delimiter status (SMSDEL,SMSDCB) as the application programs are called and released.

Any 1-byte value may be specified as a delimiter. Printable characters, however, should be avoided because they may appear in a field, and the controller would mistake data for delimiters. Also, if the field will be transmitted to one of the terminals, values from hex 00 to hex 3F should not be used because they are reserved for control characters.

Name	Operation	Operand
label	DEFDEL	(char,mask) , (char,mask),...

char

Is any byte of data to be recognized as a delimiter. The maximum number of characters allowed is 255.

mask

Is any 1-byte data expression. The bit configuration created is used as the mask.

The char and mask operands may be expressed as hexadecimal (X'*nn*'), character (C'*c*'), or binary (B'*nnnnnnnn*') data.

Note: The label of an EQUATE instruction *cannot* be used as an operand for the DEFDEL instruction.

Programming Notes: Delimiters can be divided into sets using the mask associated with each delimiter; all delimiters in a set have the same mask. Delimiter sets are used when several types of operations are performed at the same keyboard. For example, the application program may support both normal transactions and an adding machine function. A period, minus sign, or slash might be an acceptable delimiter during normal processing, but might be data when the teller is using the adding machine function. A field in Segment 1, SMSDCB, is set by the application program to indicate which delimiters are in use; if any corresponding bits in the SMSDCB and the instruction mask are set (=1), the delimiter is recognized by the controller.

Assume, for example, that three delimiters are being used, and the following masks have been associated with them (the values used as delimiters are not shown; they are referred to as 1, 2, and 3):

Delimiter	Mask
1	X'01'
2	X'02'
3	X'80'

If SMSDCB has been set to X'01', only delimiter 1 is recognized. (The result of ANDing the value in SMSDCB and the mask associated with delimiter 1 is not 0.) If SMSDCB has been set to X'03', both delimiter 1 and 2 are recognized. If SMSDCB has been set to X'82', both delimiter 2 and 3 are recognized. If SMSDCB has been set to X'FF', all delimiters are recognized.

SMSDCB is set in two ways:

- By specifying the initial value in the DELSET operand of the STATION configuration macro instruction.
- By instructions in the application program.

The initial value is set for each station when the controller is loaded and altered only by the application program.

To speed up the scan for delimiters, the DEFDEL instruction can be used to create a number of delimiter tables, each containing a relatively few entries. You must then load the address of the desired delimiter table into the SMSDEL field or a register number into the SMSDRG field in Segment 1. With the use of register addressing, a delimeter table may be found in segments other than 14, as well as in segment space different from that of the application referring to it. Under application program control, the tables can be switched as desired. If this approach is taken, specification of a delimiter table address must not be made in the BEGIN instruction.

When SETFPL and SETSFP scan for a delimiter, each character in the field is checked against the one-byte values in the delimiter table. The checking procedure used by the controller changes, depending on whether the table contains multiple specifications of a single one-byte value or the table consists entirely of unique one-byte values. The DEFDEL instruction will determine whether there are multiple specifications of a single one-byte value when the DEFDEL instruction is assembled.

If the delimiter table contains multiple specifications of a single one-byte value, the controller will check each one-byte value until a match is made between the character in the field and the value in the table, or until the end of the table is reached. If a match is found, the controller will check for a valid mask. If the mask is not valid, the controller will resume checking the delimiter table until another match is found, so that the mask can be checked, or until the end of the table is reached. If the mask is valid, the controller will set the appropriate field pointer. Any time the end of the delimiter table is reached, the controller begins processing the next character in the field.

If the delimiter table consists of unique one-byte values, checking proceeds in the same manner as it does for tables with multiple specifications of a single one-byte value until a match is found between a character in the field and the one-byte value in the delimiter table. If the mask is not valid, the controller immediately begins processing the next character in the field without checking the remaining one-byte values in the delimiter table.

Any one byte value may be specified as a delimiter. Printable characters, however, should be avoided because they may be within a field and the controller could mistake them for delimiters. Also, if the field will be read from or written to a terminal then the hex values from 00 to 3F should **not** be used because they are reserved as control characters.

DEFDMP--Define APBDUMP Buffer

DEFDMP defines the constants necessary for the APBDUMP instruction and defines a segment for APBDUMP to use as a buffer for its processing. DEFDMP is required if APBDUMP is used, and should be issued only once. It must be issued before APBDUMP, but after COPY DEFSMS. It should be coded with the data definition instructions of the controller application program, and must occur before decimal location 4095 in Segment 14.

Name	Operation	Operand
[label]	DEFDMP	SEG=n

n

Is the number of the segment (2-12, or 13) that APBDUMP can use as a buffer during its execution. The buffer must be greater than, or equal to, 454 bytes and must begin at location 0 in the segment.

Note: The data in this buffer area before APBDUMP is issued is lost when APBDUMP is executed; thus, if Segment 15 is specified, the data in the global machine section (GMS) is lost.

DEFLD--Define Field

DEFLD associates a label with an area of segment storage. This area can then be referred to in other instructions without altering the field pointer or length indicator of the actual segment. DEFLD, however, must be coded before any of the other instructions that refer to the field it defines.

Note: The maximum number of DEFLDs that may be included in an assembly is 4095.

Name	Operation	Operand
[label]	DEFLD	seg, [disp | (abs exp) | *] , {len | (abs exp)} [,BDY=HALF]

seg

Is the number (0-15) of the segment in which the field is defined, or is the label of an EQUATE instruction that has a value of 0-15.

disp

Is the location of the field within the segment or the label of a previous DEFLD or DEFCON instruction. If the label of a DEFLD or DEFCON instruction is used, the location value assigned to this DEFLD instruction is the same as that specified in the instruction indicated by the label. If the *disp* operand is omitted, the value assumed is 0 for the first occurrence of a DEFLD instruction referring to a particular segment, or is the sum of the location and length of the last DEFLD instruction that referred to the segment.

Note: If RELOC=Y is specified in the APOPT instruction, specifying a label in an external CSECT will result in an error.

abs exp

Is an absolute expression of the location of the field within the segment. When used in place of length, it is an absolute expression of the length of the defined field. The absolute expression may be:

1. An absolute value.
2. The label of an EQUATE instruction.
3. The label of another DEFLD, a DEFRF, or DEFCON instruction.
4. The displacement attribute of a DEFLD, a DEFRF, or a DEFCON; by specifying the label of the DEFLD, DEFRF, or DEFCON as follows:

   D:label

5. The length attribute of a DEFLD, DEFRF, or DEFCON instruction; by specifying the label of the instruction as follows:

   L:label

6. An arithmetic expression using the above five expressions in the following forms:

   a. (abs exp + abs exp)
   b. (abs exp – abs exp)
   c. (abs exp * abs exp)
   d. (abs exp/abs exp)

When the label of a DEFLD, DEFCON or DEFRF is used without an attribute modifier, the value of the expression is the displacement of the field when used in the displacement operand, and is the value of the field length when used in the length operand.

When the label of an EQUATE instruction is used, the value of the EQUATE must be an absolute number.

*****

Indicates that the field is to begin at the highest displacement that has been defined by previous DEFLD's for this segment.

Note: The value of disp, the absolute expression, or the sum value of the last DEFLD location plus its length must not exceed 65 535 bytes.

len

Is the length of the defined field (0 to 65 535). The length must agree with any length limitations set by instructions that refer to this DEFLD.

BDY=HALF

Aligns the field on a halfword boundary. When the specified displacement is not on a halfword boundary, BDY=HALF aligns the field on the next halfword boundary. If this operand is omitted, the field is aligned on a byte boundary only.

Programming Notes:

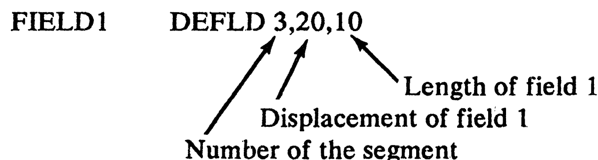
- When you are coding the instruction

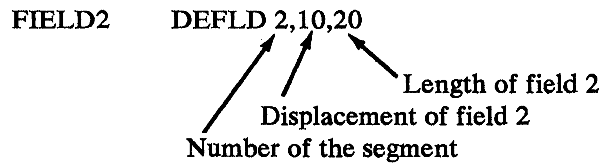
  [label] DEFLD seg,[disp],len

  you should omit the *disp* operand or it should be the label of a previous DEFLD or DEFCON. The disp operand is used to redefine fields within previously defined fields.

- When the assembler language instruction format provides for the choice *defld* or *seg,disp[,len],* use *defld.* This may require the addition of a DEFLD instruction to define *seg,disp,len,* if the DEFLD does not already exist.

Using DEFLD you could define fields as follows:





DEFLD can be used to define overlapping fields. For example:

```
FIELD1  DEFLD  3,0,20
FIELDS  DEFLD  3,10,20
```

DEFLD will calculate the displacement of the next sequential field if a displacement is not specified. For example:

```
FIELD1  DEFLD  3,0,10
FIELD2  DEFLD  3,,10
```

The displacement of the beginning of FIELD2 is set to the first byte after the end of FIELD1.

DEFRF--Define a Modified Register Address Field

This instruction associates a 'label' with a segment, displacement, and length for modified register addressing. The most DEFRF instructions that your program can contain is 4095.

Name	Operation	Operand
label	DEFRF	reg, [disp | (abs exp) | (label)] , { len|(abs exp) }

reg

Is the register (0-15) that contains the register address.

disp

Is the displacement (0 - 65 535) added to the register address to create a modified register address that locates the specified field.

abs exp

Is an absolute expression of the location of the field within the segment. When used in place of length, it is an absolute expression of the length of the defined field. The absolute expression may be:

1. An absolute value.
2. The label of an EQUATE instruction.
3. The label of another DEFLD, a DEFRF, or DEFCON instruction.
4. The displacement attribute of a DEFLD, a DEFRF, or a DEFCON; by specifying the label of the DEFLD, DEFRF, or DEFCON as follows;

   D:label

5. The length attribute of a DEFLD, a DEFRF, or DEFCON instruction; by specifying the label of the instruction as follows:

   L:label

6. An arithmetic expression using the above five expressions in the following forms:

   a. (abs exp + abs exp)
   b. (abs exp – abs exp)
   c. (abs exp * abs exp)
   d. (abs exp/abs exp)

When the label of a DEFLD, DEFCON, or DEFRF is used without an attribute modifier, the value of the expression is the displacement of the field when used in the displacement operand, and is the value of the field length when used in the length operand.

When the label of an EQUATE instruction is used, the value of the EQUATE must be an absolute number.

Note: The value of disp, the absolute expression, or the sum value of the last DEFLD location plus its length must not exceed 65 535 bytes.

label
Is the label of another DEFRF statement that defines a displacement value.

len
Is the field length (0 - 65 535).

Programming Notes: DEFRF allows you to assign a label to a field selected by a register address. This allows you to use the same register address to define a series of contiguous fields, such as in dummy sections (DSECTs), and then refer to any of the fields symbolically using the label assigned by one or more DEFRF instructions.

When a DEFRF label is used as an instruction operand, the segment is obtained from the register address. The displacement specified by the DEFRF instruction is added to the register address displacement, and the DEFRF length replaces the length from the register address.

The following example shows how DEFRFs can be used:

Assume the following record definition is repeated a total of 10 times and is located in Segment 2 beginning at displacement 47:

two-byte record type field (TYPE):
one-byte flags field (FLAGS):
two-byte record code field (CODE):
one-byte data ID:
fourteen-byte data field (DATA):

Thus, the record length is 20 bytes and can be represented with the following instructions:

```
RECORD   DEFRF   4,0,20          This entry refers to the entire record.
TYPE     DEFRF   4,(RECORD),2    First entry is length 2 and is based on
                                 the starting location of the record
                                 (disp=0).
FLAGS    DEFRF   4,,1            Second entry is length 1 and
                                 immediately follows TYPE.
CODE     DEFRF   4,,2            Third entry is length 2.
ID       DEFRF   4,,1            Fourth entry is length 1.
DATA     DEFRF   4,,14           Fifth entry is length 14.
```

Initialize the register by using the instruction LDRA (Load Register Address). In this example, the LDRA instruction would look like:

```
         LDRA    4,2,47,20       Load the current segment space ID,
*                                then load the base address of
*                                Segment 2, displacement 47 and
*                                record length 20 into register 4.
```

You can move the starting address of any record by adding to, or subtracting from, the base register, the multiple of the record length. As long as the base register address increases or decreases only by a multiple of the record length, the symbol TYPE always refers to the type of one of the 10 records, FLAGS always refers to the flags field in that same record, and so on.

The length value loaded into a register is only significant if the normal (reg) form of register addressing is to be used. In this case, the length of the operation is determined by the length in the register. For modified register-addressed fields, the length of the operation is determined by the length associated with the symbol and the length in the register is ignored.

DEFSTOR--Define Segment Storage

Segment space for the work station's initial application program is defined during either configuration or while the application program is assembled, but (except for Segment 13) is not allocated until the program is loaded. A called application program can either define and allocate its own storage, or it can share storage with the calling application program.

The DEFSTOR instruction allows you to define segment storage in your application program. You can define Segments 0 and 2 through 13 with DEFSTOR. The DEFSTOR instruction must be within the first 4095 program bytes. If the station configuration and DEFSTOR define the same segment, the DEFSTOR values override the configuration values. DEFSTOR creates a table of segment sizes that follows the FINISH instruction in the program listing.

Name	Operation	Operand
[label]	DEFSTOR	SEGSIZE=({seg0 | *} , {seg2 | *} ,...,{seg13 | *})
		[,MAXSTOR=n] [,USE={STATIC | DYNAMIC}]
		[,ID=n]

SEGSIZE
Are decimal values that specify the length, in bytes, of the segment or segments. The seg0 value defines the user portion of Segment 0 only; exactly 96 additional bytes are allocated for registers. Specifying an asterisk (*) causes the segment space to be determined by the DEFLD and DEFCON statements defined for that segment. The SEGSIZE parameters are positional, and intervening commas (,) must be coded for segment definitions you omit.

Notes:

If CPGEN or the primary application program defined a Segment 13 for any logical work station using this program, the following rules apply:

1. You can override the CPGEN definition when the program defined in the CPGEN is sharing the same storage class as your program and no other program defined in the CPGEN is sharing that class for Segment 13.

2. When the CPGEN has not declared a shared Segment 13 for the stations to which the application is assigned, then a shared Segment 13 can be assigned via the DEFSTOR.

3. A global Segment 13 can be assigned only by CPGEN and the STARTGEN instruction, and when declared it is assigned to all stations that have not been assigned a shared 13. If the program overrides the global Segment 13, then it will have a shared Segment 13 for those stations to which the program is assigned. The global 13 remains with those programs that do not have a shared Segment 13.

Failure to follow these rules, which apply to all logical work stations using your program, may cause an error and end the IPL procedure.

MAXSTOR

Used by the primary application program to define the total storage requirements for the logical work station. MAXSTOR defines the total segment storage assigned to the station in bytes; this value overrides the MAXSTOR value defined on the STATION macro. Specify *n* as a decimal value.

USE

If your program will be called more than once by another application program, specify USE=STATIC if you want the using station to hold the storage space needed by your program between calls. This storage remains unchanged until your program is recalled. If you do not specify USE=STATIC, the space is classified as DYNAMIC (the default), and released for use by other application programs in this work station.

ID

Is the identification of an application storage pool. Specify *n* as a decimal value between 1 and 15. The default value is 1.

Programming Notes: You must include a DEFSTOR instruction in your program if you specify the APENTRY operand of the BEGIN instruction.

DEFSTOR will define a new Segment 0 if you indicate that no segments are to be defined and specify USE=STATIC. If DEFSTOR is coded in this manner an additional 96 bytes must be included in the MAXSTOR value for the new segment.

DIVFLD--Divide Field

DIVFLD algebraically divides the binary contents of a register by the binary contents of a field. The length of the field must be 6 bytes or less. The quotient is placed in the register, and the remainder is lost. A divisor of 0 results in no operation.

Name	Operation	Operand
[label]	DIVFLD	reg1, { defcon2 / defld2 / (defrf2) / (reg2) / seg2,disp2,len2 }

```
                                 ⎧ defcon2          ⎫
                                 ⎪ defld2           ⎪
[label] DIVFLD    reg1,          ⎨ (defrf2)         ⎬
                                 ⎪ (reg2)           ⎪
                                 ⎩ seg2,disp2,len2  ⎭
```

operand 1
Is a register that first contains the dividend, and later the quotient.

operand 2
Is a field containing the divisor. The length of the field must be from 0 to 6 bytes; if the length is 0, no operation takes place.

Condition Codes: One of the following is set.

Hex Code	Possible Mnemonic	Explanation
01	Z0	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	ZD	The divisor is 0.

Program Checks (hex): 01, 02, 03, or 27 can be set.

DIVFLDL--Divide Field Logical

DIVFLDL divides the contents of a register by a 6-byte field. If the field is less than 6 bytes in length, it is treated as a 6-byte field by propagating zeros. DIVFLDL then divides the binary contents of a register by the binary contents of a field. The quotient is placed in the register and the remainder is lost. A divisor of 0 results in no operation.

Name	Operation	Operand

```
                                 defcon2
                                 defld2
[label] DIVFLDL   reg1,        { (defrf2)          }
                                 (reg2)
                                 seg2,disp2,len2
```

operand 1
Is a register that first contains the dividend, and later the quotient.

operand 2
Is a field containing the divisor. The length of the divisor is from 0 to 6 bytes. If a length of 0 is specified, no operation takes place.

Condition Codes: One of the following is set.

Hex Code	Possible Mnemonic	Explanation
01	Z0	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	ZD	The divisor is 0.

Program Checks (hex): 01, 02, 03, or 27 can be set.

DIVREG--Divide Register

DIVREG algebraically divides the binary contents of one register by the binary contents of another register. After the division, the quotient replaces the dividend and the remainder replaces the divisor. Both the quotient and remainder have the same sign.

Name	Operation	Operand
[label]	DIVREG	reg1, reg2

operand 1
Is a register that contains the dividend. At the end of the operation, the quotient replaces the dividend.

operand 2
Is a register that contains the divisor. At the end of the operation, the remainder replaces the divisor.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	ZD	The divisor is 0.

Program Checks: None are set.

DIVZ--Divide Zoned Decimal

This instruction divides the zoned decimal operand 1 by the zoned decimal operand 2, and replaces operand 1 with a zoned decimal result. The length of either operand is 1-63 bytes; an operand longer than 15 bytes must be selected using register addressing.

Note: This is an optional instruction; module P31 must be specified on the OPTMOD configuration macro.

Name	Operation	Operand
[label]	DIVZ	{ defld1 | (defrf1) | (reg1) | seg1,disp2,len1 } , { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
: Is a field containing the zoned decimal dividend, and the location of the result (quotient); any remainder is lost. If the result is shorter than the operand length, each high-order result byte is set to X'FO'. The field must not be in Segment 14.

operand 2
: Is a field containing the zoned decimal divisor. If zero, the appropriate condition code is set, and the operation ends.

Condition Codes: The following can be set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	ZD	Divisor is 0.

Program Checks (hex): 01, 02, 03, 09, or 27 can be set.

DTACCESS--Data Access

DTACCESS allows the accessing of application programs, as data for those programs that have been put in the SYSAP data set on the operating diskette. These "programs" would typically contain non-executable instructions such as DEFCONs. Addressing of the program "segment" is accomplished through register addressing. The program may be either resident or transient. This function allows data to be handled by the Host Transmission Facility and the System Monitor functions to add, delete, or update single application programs. This assumes that the file has a standard program header. The station, under which the program issuing the DTACCESS is operating, is put into a wait state until storage becomes available and the data program has been loaded. When the program has been successfully loaded, the station will be removed from the wait state. If the requested program is resident, DTACCESS will return the address of the resident program in register 1.

Assuming sufficient storage space is available to accommodate the data programs, a station may have up to 11 outstanding requests at a given time. DTACCESS will load the requested application program and return its address, in LDRA format, in register 1. If DTACCESS determines that the requested program has already been loaded, it simply returns the address of the requested program in register 1.

Your configuration specifications must include several macros and operands in order for you to use the DTACCESS (and DTAFREE) instruction. See APLIST, STATION, and TRANPL in the *IBM 4700 Programming Library: Volume 6*.

The format of this instruction is:

Name	Operation	Operand
[label]	DTACCESS	{ defld2 | (defrf2) | (reg2) | seg2,disp2, } [, WAIT= { Y|N }]

(Y is underlined as the default.)

operand 2
Is a field containing the parameter list. The format of this parameter list is:

bytes 0-7 = apname --> 8-byte name of the data file

Condition Codes: One of the following is set.

Hex Code	Possible Mnemonic	Explanation
01	OK	Successful execution.
02		DTACCESS canceled via Attention or because data could not be loaded from diskette.
04		No pool block available (NOWAIT).
08		Requested data already accessed.

Program Checks (hex): 01, 02, 21, 25, 27, or 28 can be set.

Hex Code	Explanation
01	The specified PLIST is in an undefined segment.
02	The PLIST length extends beyond the segment.
21	Requested data not found.
25	No room in segment header area (DTACC too small).
27	DTACC=0 specified (seg space id invalid).
28	Target data is transient but no pool is defined.

Programming Notes: The application program that issues this instruction may put its work station into the wait state until storage becomes available. See the APCALL instruction for further information.

DTAFREE--Data Free

DTAFREE will free application programs accessed as data by the DTACCESS instruction.

If the application program to be freed is transient, the storage area containing this program is made available for other uses. The segment header associated with an application (resident or transient) that has been freed is also available for use.

Name	Operation	Operand
[label]	DTAFREE	{ defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 2
Is a field containing the parameter list. The format of the parameter list is:

bytes 0-7 = apname --> 8-byte name of the data file

Condition Codes: The following may be set.

Hex Code	Possible Mnemonic	Explanation
01	OK	Successful execution.

Program Checks (hex): 01, 02, and 27 can be set.

EDIT--Edit Monetary Field

EDIT formats a field by suppressing leading zeros, adding fill characters, and adding monetary symbols. The mask that EDIT uses in formatting a field is created by the MASK instruction, and is made up of the following characters:

fill

Is a 1-byte, hexadecimal, character, or binary expression used in place of leading zeros when zero suppression is specified.

9

Indicates significant position. A number from the field to be edited that corresponds to this position in the mask is moved to the field that is to contain the edited data without inspection.

Z

Indicates the leading zero suppression. Leading zeros in the field to be edited are suppressed, and the corresponding position in the field to contain the edited data is loaded with a fill character. Characters from the field to be edited, other than leading zeros, that correspond to this position in the mask are moved to the field that is to contain the edited data.

$

Indicates a dollar-sign insertion. Significant digits are inserted in the edited field. When the last leading 0 is encountered, the dollar sign is inserted in place of the 0 and all leading zeros are replaced by the fill character specified in the MASK instruction. If, however, (1) the next character in the field to be edited is not a leading 0 and the next mask character is not an insertion character, or (2) the next mask character is 9 or Z, "$", together with all other characters in the field to be edited, is moved to the field that is to contain the edited data.

b

Indicates blanks insertion. Blanks in the mask are always inserted in the corresponding positions of the field that is to contain the edited data.

Any other characters except 9, Z, $, and b may be insertion characters. They are inserted in the corresponding positions of the field that is to contain the edited data after a significant digit has been included in this field. (See the MASK instruction for a further description of the mask.) If no significant digit is found in the field to be edited before the last insertion character, all insertion characters are replaced by the fill character.

If the field to be edited is longer than the mask, the field is truncated on the left. If the field to be edited is smaller than the mask, the edited field is padded with zeros or fill characters, as specified in the MASK instruction. Note that the length of the mask does not include insertion characters.

Following the execution of EDIT, the primary field pointer in the segment containing the edited field points to the first character after the last character moved. The field length indicator is unchanged.

Name	Operation	Operand
[label]	EDIT	seg1,seg2, {label3 | (reg3)}

operand 1
Is a field in the specified segment to place the edited data. The field's location is indicated by the primary field pointer (PFP), and the field length indicator is ignored. If seg1 and seg2 specify the same segment, the field is indicated by the secondary field pointer.

operand 2
Is a field in the specified segment to be edited. The field's location is indicated by the primary field pointer and the length by the field length indicator.

label 3
Is the label of the MASK instruction defining the mask to be used during editing. The length of the edited field in seg1 is determined by the length of the mask. 'label' may be an external symbol when RELOC=Y is specified in the APOPT instruction.

(reg3)
Is a field containing a mask (in the same format as the data generated by the mask instruction). The length of the edited field in seg1 is determined by the length of the mask. The length specified in reg3 is ignored.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	Editing was successful.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, or 0E can be set.

Programming Notes: For example, the message received from the host processor contains the transaction amount and the new balance as binary numbers without monetary symbols or decimal points. The following example shows the instructions that edit the new balance before printing the customer's passbook. The field transmitted from the central processor is:

03232E

Its EBCDIC equivalent is:

0000205614

The edited field is:

bbbb$2,056.14

```
AMNTMASK  MASK    C'$$,$$$,$99.99'                 1
CPTOTAL   DEFLD   CPINSEG,22,3                     2
WORKAREA  DEFLD   WORKSEG,,10                      3
PRNTOUT   DEFLD   PRNTSEG,15,13                    4


EDITAMNT  LDFLD   REG1,CPTOTAL                     5
          STFLDC  REG1,WORKAREA                    6
          SETFPL  WORKAREA                         7
          SETFPL  PRNTOUT                          8
          EDIT    PRNTSEG,WORKSEG,AMNTMASK         9
```

1 Indicates that at least four digits and a decimal point must always appear in the output.
A monetary sign will also appear if the field to be edited contains fewer than 10 characters.
When leading zeros are encountered in the leftmost portion of the data, one monetary sign is inserted and the edited field is padded to the left with blanks. If the mask was defined as MASK C'*',C'$ZZ,ZZZ,Z99.99' the edited field would appear as $*****2,056.14.

2 Defines the field that contains the binary value transmitted from the central processor.

3 Defines a workarea used when the field is converted from binary to EBCDIC.

4 Defines the location of the result.

5 Loads the binary amount field into a register.

6 Stores the EBCDIC equivalent of the register contents into the work area.

7 Sets the PFP and FLI of WORKSEG to define the EBCDIC amount in WORKAREA.

8 Sets the PFP and FLI of PRINTSEG to define the location of the edited field.

9 Edits the total and places it in the output area.

ENDINIT--End Initialization Section

This instruction defines the end of an initialization section of the program that begins with an SINIT instruction, continues with INITSEG instructions, and ends with an ENDINIT instruction. No other 4700 Assembler instructions can be within this sequence.

Name	Operation	Operand
[label]	ENDINIT	

ENDOVLY--End of Overlay Section

ENDOVLY specifies the end of an overlay section. It must be the last instruction in the overlay section.

Name	Operation	Operand
[label]	ENDOVLY	

ENDSEG--End Application Program Section

ENDSEG indicates to the assembler the end of a controller application section that was begun with a SEGCODE instruction. ENDSEG is a required instruction in application sections that start with SEGCODE.

Name	Operation	Operand
	ENDSEG	

Note: The assembler instruction END must be used to end an assembly.

EQUATE--Equate a Label to a Value

EQUATE associates a label with a character or a value. This label can then be used in other instructions in place of the value or character. EQUATE, however, must be coded before any of the other instructions that refer to the label.

Note: The maximum number of EQUATEs that may be used in an assembly is 4095.

Name	Operation	Operand
label	EQUATE	{ value / (abs exp) }

value

Is any valid decimal, hexadecimal, binary, fullword, halfword, address, or character specification.

abs exp

Is an absolute expression of the value to be equated. The absolute expression may be:

1. An absolute value.
2. The label of an EQUATE instruction.
3. The displacement attribute of a DEFLD, DEFRF, or a DEFCON; by specifying the label of the DEFLD, DEFRF, or DEFCON as follows:

   D:label

4. The length attribute of a DEFLD, a DEFRF, or a DEFCON; by specifying the label of the DEFLD, DEFRF, or DEFCON as follows:

   L:label

5. An arithmetic expression using the preceding four expressions in the following forms.

   a. (abs exp + abs exp)
   b. (abs exp – abs exp)
   c. (abs exp * abs exp)
   d. (abs exp/abs exp)

Note: The maximum length of value is 8 characters in DOS/VS and 255 characters in OS/VS.

Programming Notes: For example, a controller application program may always use Segment 2 as the input area and register 3 to contain the total amount of the transaction. The following could be coded:

```
INPTAREA  EQUATE  2
TRANTOT   EQUATE  3
```

The labels INPTAREA and TRANTOT would then be used in applicable instructions rather than the absolute numbers. To change the input segment number, change only the value specified in the EQUATE. When the program is assembled again, the new value will appear in applicable instructions.

ERRLOG--Obtain Statistical Counters

ERRLOG causes the device type, station ID, and statistical counters for a particular component to be stored in a segment. The statistical counters include both error counters (for example, machine failures) and diagnostic counters (improper operating procedure).

After instruction operation, the PFP still points to the start of the field containing the stored information, and the FLI contains the field length. ERRLOG returns the modulus value in the last four bits of the device address for loop terminals, and the loop speed and clocking indicator in the last eight bits of the physical loop address. For device cluster adapter (DCA) terminals, the last four bits of the physical device address are set to binary zeros.

Name	Operation	Operand
[label]	ERRLOG	seg1, { defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 1
Contains the device type code, station ID, and statistical counters. The segment number cannot be 14. The location of the field, within the segment that is to contain the statistical counters, is indicated by the PFP. The FLI is ignored. At the end of the operation the PFP still points to the beginning of the field, and the FLI contains the field length. The format of the information returned is explained in Figure 5-1 on page 5-120.

operand 2
Defines the field containing the physical device address of the component for which the statistical counters are to be stored. The format of this address is explained in Figure 5-2 on page 5-121. The length associated with this operand is ignored, and the first 2 bytes are assumed to contain the physical device address.

Note: The label of a DEFCON cannot be used.

Condition Codes: One of the following is set:

Hex Code	Explanation
01	The instruction was executed successfully.
02	There was an invalid device specification.
04	There was insufficient space in the segment to store the counters.

Program Checks: 01, 02, or 27 can be set.

Code

Is a 1-byte component code, as follows:

Code	*Component*
X'01'	Communication Link
X'02'	Diskette
X'03'	ALA Line
X'04'	Disk
X'05'	Encryption Facility
X'80'	Loop
X'81'	4704/3604/3278/3279 Keyboard Displaywriter Personal Computer
X'82'	4704/3604/3278/3279 Display
X'83'	3610/3612 Document Printer; 3611/3612 Passbook Printer
X'84'	3262/3287/3289 Printer
X'86'	4707/3604 Magnetic Stripe Encoder
X'87'	3614/3624 Consumer Transaction Facility
X'88'	3606/3608 Keyboard/Display
X'89'	3608 Printer
X'8A'	3615 Administrative Terminal Printer
X'92'	3616 Printer
X'95'	Device Cluster Adapter
X'9A'	4710 Printer
X'AB'	Magnetic Stripe Encoder for 4704 Models 2 and 3
X'B0'	4720 Printer

Station ID

Is the binary ID number of the station associated with the component.

Statistical Counters

Are a variable number of counters.

Figure 5-1. Format of Statistical Counters Returned by ERRLOG

Primary Diskette	x'9'	X'0'	X'2'	X'0'
Auxiliary Diskette	X'9'	X'0'	X'3'	X'0'
Disk A	X'9'	X'0'	X'7'	X'0'
Disk B	X'9'	X'0'	X'8'	X'0'
Host Link	X'9'	X'0'	X'1'	X'0'
Loop	Loop	X'0'	X'0'	X'0'
Loop Device	Loop	Terminal	Component	X'0'
DCA Adapter	X'9'	X'A'	X'0'	X'0'
DCA Port	X'A'	Port	X'0'	X'0'
DCA Device	X'A'	Port	Component	X'0'

Loop

Is the 4-bit binary loop number (1-4) assigned during the controller configuration procedure.

Terminal

Is the 4-bit binary terminal address, established at the terminal by setting address switches on the terminal itself.

Port

Is the 4-bit binary number (0-7) of the DCA port to which the terminal is connected to the controller.

Component

Is the 4-bit component address of a terminal component, as follows:

Component Address	Component
0001	4704/3604/3278/3279 Keyboard
0010	4704/3604/3278/3279 Display
0011	4704/3604 Magnetic Stripe Encoder/Reader
0100	4710/3610/3612 Document Printer
0101	3611/3612 Passbook Printer
0100	3615 Administrative Terminal Printer (not address shared)
0110	3606/3608 Keyboard/Display
0111	3608 Printer
1000	3614/3624 Terminal
0100	3262/3287 Printer
nnnn*	3615 Administrative Terminal Printer (address shared)
nnnn*	3616 Journal Printer
nnnn*+1	3616 Document Printer

*nnnn = setting of the units subaddress switches

Figure 5-2. Physical Device Address Used by ERRLOG

EXOR--Exclusive OR

EXOR performs an exclusive OR operation of two fields, and places the result in operand 1. When the exclusive OR is executed, a 0 is set in the result if corresponding bits in the tested fields are alike; a 1 is set in the result if corresponding bits in the tested fields are not alike. The length of the operation is governed by the length of the second field; this length must be in the range of 1 to 255 bytes.

Name	Operation	Operand
[label]	EXOR	{ defld1 | (defrf1) | (reg1) | seg1,disp1 } , { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp,len2 }

operand 1
: Is a field that is to be exclusive-ORed with operand 2, and is to contain the result. The length specification for this field is ignored; the length of the operation depends on the length of operand 2. The field must not be in Segment 14.

operand 2
: Is a field used in the exclusive OR operation with operand 1.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is all zeros.
02	NZ	The result is mixed ones and zeros, or all ones.

Program Checks (hex): 01, 02, 03, or 27 can be set.

EXORI--Exclusive OR Immediate

EXORI performs an exclusive OR operation of a 1-byte or 2-byte field with a 1-byte or 2-byte immediate operand, and places the result in operand 1. When the exclusive OR is executed, a 0 is set in the result if corresponding bits in the tested fields are alike; a 1 is set in the result if corresponding bits in the tested fields are not alike.

Name	Operation	Operand
[label]	EXORI	{ defld1 | (defrf1) | (reg1) | seg1,disp1 } , immdata2

operand 1
Is a field that is to be exclusive-ORed with operand 2, and is to contain the result. The length specification for this field is ignored; the length of the operation depends on the length of operand 2. The field must not be in Segment 14.

operand 2
Is 1 or 2 bytes of immediate data.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is all zeros.
02	NZ	The result is mixed ones and zeros, or all ones.

Program Checks (hex): 01, 02, 03, or 27 can be set.

EXPS--Exchange Primary and Secondary Field Pointers

EXPS swaps the primary and secondary field pointers of the indicated segment.

Name	Operation	Operand
[label]	EXPS	seg1

operand 1
Is the number of the segment for which the pointers are to be swapped.

Condition Codes: The code is not changed.

Program Checks (hex): 01 can be set.

FCLENTER--Define COBOL Entry Linkage

You can use FCLENTER at entry of a program to assist in following the COBOL linkage conventions. On entry, register 1 contains the register address of a list and the list contains register addresses that locate the parameters. The called program can, using FCLENTER, save the address of the list (that is; the value in register 1) and move the parameters to fields in the current segment space.

FCLENTER uses registers 8 and 9 as work registers.

Name	Operation	Operand
[label]	FCLENTER	saveaddress [,a1,a2,...,a254]

saveaddress
Is the label of a DEFLD instruction where the address of the list is stored. The length of the field should be at least 6 bytes. This operand must not be in Segment 14. Register 1, which by COBOL convention always contains the parameter list address, is stored at this location.

a1,a2,...,a254
Are labels of DEFLD instructions defining the target locations of the parameter move operations. These fields must not be in Segment 14. These names identify the parameters in sequence as they occur in the COBOL program's parameter list. You may define up to 254 parameters on an FCLENTER instruction.

Condition Codes: This instruction may modify the condition code; however, any condition code returned will have no significance.

Program Checks (hex): 01, 02, 03, 09, 11, and 27 may be set.

Programming Notes: FCLENTER and FCLEXIT are used in conjunction, FCLENTER at the entry point of the assemler program being invoked, and FCLEXIT at the point of return to the COBOL program.

FCLEXIT--Define COBOL Exit Linkage

FCLEXIT can be used at the return logic of a program to assist in following the COBOL linkage conventions; register 1 is to in following the COBOL linkage conventions; register 1 is to contain the register address of a list, the list contains register addresses which locate parameters. The called program can, using FCLEXIT, set register 1 to the address of the list, move the parameter from the fields specified by a1,a2,... (below) to the locations specified in the list, and then APRETURN.

FCLEXIT uses registers 2 and 3 as work registers.

Name	Operation	Operand
[label]	FCLEXIT	saveaddress [,a1,a2,...,a254]

saveaddress

If the label of the DEFLD or DEFCON instruction that contains the address (of the list) to be loaded into register 1. If your program received parameters from the calling COBOL program, *saveaddress* is the same as that coded on the FCLENTER instruction.

a1,a2,...,a254

Are labels of DEFLD instructions defining the source locations of the parameter move operations. If your program received parameters from the calling COBOL program, the parameter names can be the same as those you specified on the FCLENTER instruction. If you use the FCLENTER names but do not return a result parameter in place of any but the last name, you must code a comma (,) in place of the missing result parameter; the result parameters in this case are positional.

Condition Codes: This instruction may modify the condition code; however, any condition code returned will have no significance.

Program Checks: 01, 02, 03, 09, 11, 26, and 27 may be set.

FINDAP--Find Application Program

This instruction provides two major functions: it determines the presence of an application program and it supplies information about dynamic storage management.

FINDAP determines if an application program is in your active, application program data set. You must set a function code and the name of the program to be found into a parameter list (see COPY DEFFAP in Appendix B). FINDAP sets a condition code to indicate whether or not the named application program was found.

This instruction also provides statistical information about storage management operation in the 4700.

You must include optional module M45 in your OPTMOD configuration specifications in order to use function codes 01 through 04.

Name	Operation	Operand
[label]	FINDAP	{ defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 2
Is a field containing the parameter list. The field must not be in Segment 14.

You must set the FAPFCN field in DEFFAP for one of the following:

Hex Code	Meaning
00	Determine the absence or presence of an application program.

Set by Program	Set by 4700
FAPAPN	FAPFLG (FAPTRN mask) FAPCAL FAPLOD FAPWAT

01 Obtain the amount of storage currently allocated to the specified application program on the specified work station. If the station ID is set to hex FF, you will get the amount of storage allocated to the specified program on all work stations. This amount of storage represents Segments 0, and 2 through 12.

Set by Program	Set by 4700
FAPAPN FAPCSH	FAPSTOR

02 Obtain the amount of storage currently allocated to the specified work station. The amount of storage represents segments 0, and 2 through 12.

Set by Program	Set by 4700
FAPCSH	FAPSTOR

03 Obtain the amount of storage reserved for the specified pool and the amount of storage in use. If the pool has not been defined then zero values are returned. If you specify a pool ID of 00 the values for the general pool are returned.

Set by Program	Set by 4700
FAPCSH	FAPTTL FAPSTOR

04 Obtain statistics for the specified pool. If the reset flag (see FAPCLR) is set, then the counters will also be reset to 0.

Set by Program	Set by 4700	
FAPFLG (FAPCLR)	FAPTTL	
FAPCSH	FAPSTQ	
	FAPSTN	
	FAPCBY	
	FAPCBN	
	FAPCUR	
	Counters:	FAPSMAX, FAPSAPC, FAPSRCL, FAPSREQ, FAPSCMB, FAPSDEC, FAPSSCN, FAPSMOV, FAPSLNG, FAPSWTC, FAPSWTS, FAPSWTQ

Condition Codes:

Hex Code	Explanation
For function code 00, one of the following is set:	
01	The program was found.
02	The program was not found.
For function code 01, one of the following is set:	
01	The program was found.
02	The program was not found or is not currently in main storage and in use.
For function code 02, one of the following is set:	
01	The station was found.
02	The station was not found
For function codes 03 and 04, one of the following is set:	
01	The storage pool was found.
02	The storage pool was not found.

Program Checks (hex): 01, 02, 11, or 27 can be set.

FINISH--End the Application Program

FINISH indicates the end of the conroller application program to the assembler. FINISH is a required instruction in a controller application program.

Name	Operation	Operand
	FINISH	

Note: The assembler instruction END must be used to end an assembly.

INITSEG--Initialize Segments

INITSEG permits an initial application program to initialize segment storage during controller load time. INITSEG allows you to:

1. Set some program data fields differently for each work station.

2. Override values set into fields of your program by configuration.

INITSEG instructions must be within an initialization section of your program that begins with an SINIT instruction, ends with an ENDINIT instruction, and contains no other 4700 assembler instructions. Your program can contain more than one initialization section, and each section can contain more than one INITSEG instruction. If multiple INITSEG instructions refer to the same field, it is set to the value contained in the last referring instruction.

Name	Operation	Operand
[label]	INITSEG	[id,] ({seg,disp|defld},data) [,...]

id

Specifies the work station to which the initialization values apply. If omitted, these values apply to any station using this program as its initial application program.

seg

Is the number of the segment to be initialized (2-13, 15, 0A, or 0B). 0A and 0B specify the Segment 0 for operators A and B on a shared station. If the station is not shared, specify 0A.

disp

Is a decimal number indicating the location of the data within the segment to be initialized. Specifying 0 with a 'seg' parameter of 0A or 0B selects the high-order byte of register 0. Specifying 0 with a seg of 15 specifies the beginning of user storage in Segment 15, because the system storage portion of Segment 15 cannot be specified.

defld

Is the label of a DEFLD instruction that defines the area to be initialized. The DEFLD instruction must be coded before this INITSEG instruction in your program.

data

Is the initializing data, specified in one of the following forms:

ddd

Is a decimal number specifying how often the constant ('xxx...x') occurs. If the constant occurs only once, do not specify ddd.

t

Defines the type of data in the constant. Specify X for hexadecimal, C for character data, F for fullword, and so on, as defined in Chapter 4.

nn

Is a decimal number specifying the length of the constant 'xxx...x'.

'xxx...x'

Is the constant data, enclosed in single quotation marks.

INOR--Inclusive OR

INOR performs an OR operation of two fields, and places the result in operand 1. When the OR is executed, a 0 is set in the result if neither of the corresponding bits in the fields is a 1; a 1 is set in the result if either or both the corresponding bits in the fields are 1's. The result is placed in the first operand location.

Name	Operation	Operand
[label]	INOR	{ defld1 | (defrf1) | (reg1) | seg1,disp1 } , { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Is a field to be ORed with the second operand, and the location that is to contain the result. The length specification for this field is ignored. The field must not be in Segment 14.

operand 2
Is a field to be ORed with the field defined in operand 1. The length of operand 2 is from 1 to 255 bytes, and determines how many bytes are included in the OR operation.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is all zeros.
02	NZ	The result is mixed ones and zeros, or all ones.

Program Checks (hex): 01, 02, 03, or 27 can be set.

INORI--Inclusive or Immediate

INORI performs an OR operation of a 1-byte or 2-byte field with a 1-byte or 2-byte immediate operand. When INORI is executed, a 0 is set in the result if neither of the corresponding bits in the fields is a 1; a 1 is set in the result if either or both the corresponding bits in the fields are 1's. The result is placed in the first operand location.

Name	Operation	Operand
[label]	INORI	{ defld1 | (defrf1) | (reg1) | seg1,disp1 } , immdata2

operand 1
Is a field to be ORed with the second operand, and the location that is to contain the result. The length specification for this field is ignored. The field must not be in Segment 14.

operand 2
Is 1 or 2 bytes of immediate data.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is all zeros.
02	NZ	The result is mixed ones and zeros, or all ones.

Program Checks (hex): 01, 02, 03, or 27 can be set.

INTMR--Interval Timer

INTMR starts, stops, reads, resets, or controls access to as many as 15 interval timers that each work station can have. INTMR selects a parameter list that, in turn, selects the particular interval timer and the function to be performed.

Except for starting and stopping, any work station can control interval timers for any other work station. A work station can start and stop only its own timers. When it is started, a timer counts intervals of time measured in seconds and fractions of seconds until the owning work station stops the timer. The timer also records the minimum interval measured, the maximum interval, the number of intervals measured, and the total time of all intervals measured until the timer is reset or deactivated by the owning or some other work station. Deactivation causes all values for that work station to be lost; all values are reset to zero, and the timer cannot be restarted until the timer is reactivated.

INTMR selects a parameter list having one of the three formats. The parameter list defines the timer, the operation performed, and can contain status flags for the timer as well as fields where the timer values, if any, are returned. INTMR must select a parameter list that is appropriate for the operation it performs. The parameter list can be defined using the DEFINT operand of the COPY instruction, or by your own definition. The two largest parameter lists, including the list defined by DEFINT, should not be in Segment 14; the smallest list can be in any segment. Refer to the example below for the parameter list formats, INTMR function codes, and their operation.

Note: To use this instruction, you must code the P2C operand on the OPTMOD configuration macro.

Using the INTMR Instruction: The INTMR instruction function depends on the function code in the parameter list that the instruction selects. INTMR can perform the following interval timer functions:

- Activate or deactivate one or all timers on its own or another work station. A timer must be activated before it can be started or stopped; controller startup sets all timers to the inactive state. Only activated timers are reported back to the host through CNM/CS.

- Start, stop, and read the last interval timed by any timer on its own work station.

- Read the longest interval, the shortest interval, the total time of all intervals, and the total number of intervals timed since the timer was last activated or reset. These values can be read for any timer on this or any other work station, and you may choose whether to reset the timer counters after they are read.

Threshold Analysis and Remote Access (TARA) uses the interval timers to measure response times; as a result, you must include the INTMR instructions in existing or new application programs at points that correctly measure the response times reported to the host site by CNM/CS. For this reason, you may be required to code INTMR instructions according to procedures defined by personnel at your host site. For example, your site might want to measure the time needed to process a transaction. In this case, you would code an INTMR instruction to start the timer just before issuing an LWRITE CP instruction, and another INTMR to stop the timer just before an LREAD CP instruction.

Another example is to measure the time needed to receive a response to your program's request to a terminal. You would start the timer just after issuing the LWRITE to the terminal, and stop the timer when the operator first entered data (just after the terminal entry point for an idle station, or after exit from a test of the LDA Attention Summary Field if that terminal's bit was set for either an idle or active station).

Name	Operation	Operand
[label]	INTMR	{ defcon2 defld2 (defrf2) (reg2) seg2,disp2 }

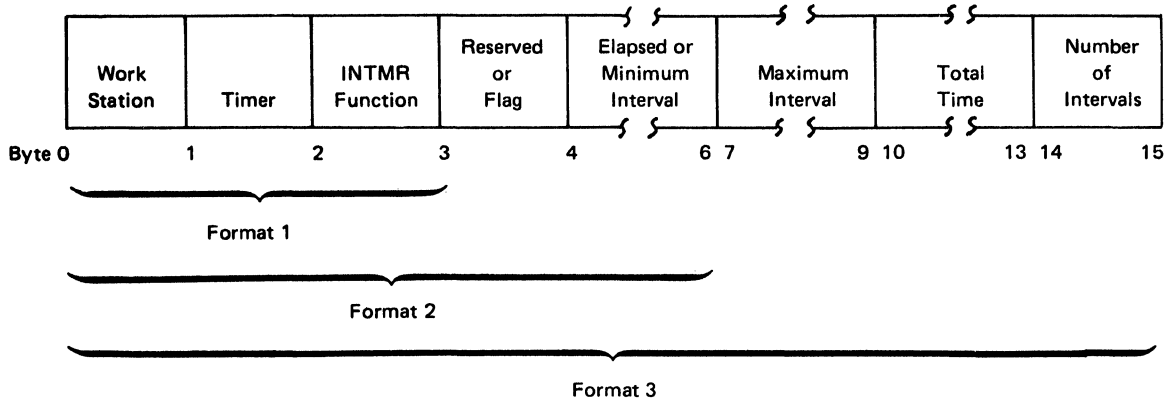
operand 2

Is a field containing the parameter list. You can define the parameter list by means of the DEFINT copy file described in Appendix B. If your program reads timers or enters data in the parameter list, the field must not be in Segment 14.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	INTMR completed successfully.
02		The specified timer is not supported (undefined) or was not activated.

Program Checks (hex): 01, 09, 11, or 27 can be set.



You must select and define a parameter list with a format appropriate for each INTMR instruction that your work station issues. Choose the format according to the functions your INTMR instructions perform. Refer to the descriptions of the INTMR function byte (byte 2) for the operations, their function codes, and format requirements that each requires.

Depending on the format chosen, specify the parameter list fields as follows:

Work Station (Byte 0)

Specify the work station owning the interval timer or timers controlled by this parameter list. This is the binary equivalent of a decimal value ranging 00 through 60. If 00 is specified, this work station is assumed. You must specify zero for all start and stop timer functions (byte 2 = X"00", X"01", or X"11").

Timer (Byte 1)

Specify the timer or timers on which the function (byte 2) is performed. Specify the binary equivalent of a decimal value ranging 1 through 15. Coding X"00" for function codes X"04" or X"05" activates or deactivates all allocated timers on the specified work station; on all other functions, X"00" causes a no-operation. Any timer specified must be within the range specified by the INTMR operand of the STATION macro, or condition code 2 is set.

INTMR Function (Byte 2)

Determines the function performed by the INTMR instruction, and the format of the parameter list. An invalid function code causes a program check of X"11" in the log message. The functions, their codes, and the minimum allowable parameter list formats are as follows:

Function	Hex Code	Description
Start Timer	00	Start the timer specified by byte 1 for this work station (byte 0 must be X"00"). The timer is not affected if it was already started. This function requires a format-1 parameter list.
Stop Timer	01	Stop the timer specified by byte 1 for this work station (byte 0 must be X"00"). If the timer is already stopped, this operation is ignored. This function requires a format-1 parameter list.
Read Timer	02	Set all interval timer results (shortest interval, longest interval, total time, and the total number of intervals) for the work station and timer specified by bytes 0 and 1. This function requires a format-3 parameter list. If there is insufficient space in the segment, INTMR causes a program check of X"11" in the log message.
Read Timer with Reset	03	Perform the same function as Read Timer (X"02"), but reset the result counters. This function requires a format-3 parameter list. If there is insufficient space in the segment, INTMR causes a program check of X"11" in the log message.
Activate Timer	04	Allow other INTMR functions to be performed on the timer or timers specified by byte 1. If byte 1 = X"00", all timers for the work station specified in byte 0 are activated (if allowed by that station's STATION macro instruction). This function requires a format-1 parameter list for each timer.

Function	Hex Code	Description
Deactivate Timer	05	Prevent other INTMR functions from being performed on the timer or timers specified by byte 1. If byte 1 = X"00", all timers for the work station specified in byte 0 are deactivated. This function requires a format-1 parameter list for each timer.
Stop Timer-- Read Elapsed Interval	11	Stop the timer specified by byte 1 for this work station, and set the last interval timed into the timer's parameter list. If the timer was not started, the interval is zero. This function requires a format-2 parameter list; byte 3 of each list is unused.

Flag (Byte 3)

For format 2, this field is unused (reserved). On INTMR instructions performing read timer or stop and read elapsed interval, the flag byte contains one, or a combination, of the following values, indicating timer status before the operation was performed.

X"00": Timer was active
X"40": Timer was running
X"80": Timer inactive at start of this operation.

Elapsed or Minimum Interval (Bytes 4 - 6)

In format 2, this field is set to the time of the interval ended by the function just performed. In format 3, INTMR sets this field to the minimum interval timed since the timer was last activated or reset. In both formats, the leftmost two bytes contain the number of seconds in the interval, and the rightmost byte contains any fraction of a second. The count stays in the parameter list until another INTMR instruction uses this parameter list, or the timer is deactivated.

Maximum Interval (Bytes 7 - 9)

INTMR sets this field to the maximum interval timed since the timer was last activated or reset. The leftmost two bytes contain the number of seconds, the rightmost byte contains any fraction of a second. The count stays in the parameter list until another INTMR instruction reads a timer using this parameter list, or until the timer is deactivated.

Total Time (Bytes 10 - 13)

INTMR sets this field to the total of all intervals timed since the timer was last activated or reset. This count ranges 0-16, 777, 215 seconds with the rightmost byte containing any fraction of a second. The count stays in the parameter list until the list is used to perform another read operation, or until the timer is deactivated.

Number of Intervals (Bytes 14 - 15)

INTMR sets this field to the total of the intervals counted since the timer was last activated or reset. The count stays in the parameter list until another INTMR instruction uses this list to perform another read operation, or until the timer is deactivated.

| IRETURN--Indexed Conditional Return

IRETURN conditionally returns control to the program that issued one of the branch-and-link instructions. It is used in conjunction with the ADRLST instruction.

Name	Operation	Operand
[label]	IRETURN	[mask|X'F'], index

mask

Is the condition to be met for the return to be performed. If any bit in the mask is set and the corresponding bit is set in the present condition code, then the condition is satisfied and the return is performed. If the condition is not satisfied, then operation continues with the next sequential instruction. If the mask is omitted, then hex F is used as the mask to cause an unconditional return.

index

Indicates the address within an address list to which the return should be made. If index is 0, then no address list exists and return is performed through the return-address stack. If index is not 0, then it is the index of the entry that contains the return address. Code index as a decimal value between 0 and 7.

Condition Codes: The code is not changed.

Program Checks (hex): 04 can be set.

Programming Notes: In the example below the program will continue operation at label CASE2.

```
         BRANL  subroutine
         ADRLST CASE1,CASE2
         .
         .
CASE1    .
         .
CASE2    .
         .


subroutine
         .
         .
         IRETURN X'F',2
```

JUMP--Short Branch

JUMP conditionally or unconditionally changes the sequence of program operation. If any bit is set to 1 in the mask specified in the instruction and the corresponding bit is set in the present condition code, then the condition is satisfied, and the branch is taken. When an unconditional branch or a branch with the condition satisfied is executed, execution continues with the instruction referred to in the JUMP instruction. Otherwise, execution continues with the next sequential instruction.

Because the JUMP instruction is only 2 bytes long, it can be used in place of BRAN (a 4-byte instruction) to save controller storage. The location branched to, however, must not be more than 510 bytes away from the JUMP instruction address.

Note: If RELOC=Y is specified on the APOPT instruction, the JUMP instruction can be used to branch only within a CSECT.

Name	Operation	Operand
[label]	JUMP	[ccmask|X'F',] branch address

ccmask
Is the condition to be met for the branch to be taken (refer to the condition codes set by individual instructions). The ccmask can be in the form of a mnemonic (see Chapter 4 for a list of the mnemonics representing coded values), a 1-byte hexadecimal expression, a 4-bit binary expression, or the label of an EQUATE instruction expressing one of the preceding numeric values. If the operand is omitted or if X'F' is coded, then the branch is always performed. If ccmask is specified as hex 0, the branch is never taken.

branch address
Is the label of the instruction where processing is to continue if the branch is taken.

Condition Codes: The code is not changed.

Program Checks (hex): 0B can be set.

LCHAP--Change Priority

LCHAP provides the application program with an instruction to control priority dispatching. When priority dispatching is ON, priorities will be defined by the tables generated by the PRIDSP configuration macro. GMSPRI contains the identification of the dispatching table being used and an indicator of priority dispatching status.

Name	Operation	Operand
[label]	LCHAP	[ON / OFF / NEXT]

ON

Specifies that priority dispatching will be active for the table indicated in bits 1-7 of GMSPRI. If LCHAP is successful, bit 0 will be switched on by LCHAP to indicate that priority dispatching is active for the table indicated.

OFF

Deactivates priority dispatching and returns control to the station chain dispatching sequence. Bit 0 of GMSPRI will be switched off.

NEXT

Will cause the station specified in SMSDSS to be the next station that the controller will attempt to dispatch. Note that priority dispatching must be active for this operand to be valid. Be careful when using this function because it is possible, through frequent selection of one set of stations, to prevent dispatching of the remaining stations.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	The change was made.
02		The change was not made. The required priority table was not defined when ON was specified; or the station id in SMSDSS was invalid; or priority dispatching was not active when NEXT was specified.

Program Checks: None are set.

Note: The dispatching status indicator in GMSPRI does not control dispatching status. It is set/reset by LCHAP for information only. Modification of this bit setting by other instructions has no effect as to whether priority dispatching is on or off. Modifications to the dispatching table ID have no effect until LCHAP ON is issued.

An application program can check the status (on or off) of priority dispatching, obtain the ID number of the priority dispatching table currently being used, and/or transfer dispatching control to another table, a one-byte field--GMSPRI--is maintained in Segment 15. The low-order 7 bits of this field contain the dispatching table ID number, and the high-order bit is used to indicate priority dispatching status. The dispatching order may be selected during program execution by loading GMSPRI with the ID of the new table, and issuing LCHAP ON. Station chain dispatching may be invoked by issuing LCHAP OFF.

LCHECK--Check Status of Station-to-Station Write

LCHECK determines the status of a station and synchronizes data transmission. Before an LCHECK is issued for another station, the station number must be stored in binary in SMSDSS. If there is an outstanding message previously written to the other station, LCHECK causes a wait state until that station has read the message.

Note: If this wait state is terminated by an operator signaling attention (SMSIND=X'40'), the outstanding write request is discarded. However, the message remains pending at the other station, and, when the LREAD is issued, a status of "no message pending" is returned. The cancel flag in SMSIND should be turned off before issuing LCHECK; if the cancel flag is on, the LCHECK will return immediately with a condition code of 02 and status (SMSDST) set to X'0800'.

When the station has completed the read operation for the write request, the condition code is set, the wait state ends, control passes to the next sequential instruction, and a status code is stored in SMSDST. If there were no outstanding write operations, a status code of 0 is returned.

Name	Operation	Operand
[label]	LCHECK	ST [,TIO]

ST
Specifies that write operations to another station are to be checked.

TIO
Specifies that the appropriate status is to be returned and the condition code set to reflect the status of the last LWRITE ST issued. The application program retains control whether the LWRITE ST has been completed or not.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	Zero status is returned.
02	ST	Nonzero status is returned: the status code is contained in SMSDST. (See Appendix E for an explanation of the status codes.)
04		The LWRITE ST has not been completed (applies only to the TIO option).

Program Checks: None are set.

LCONVERT--Convert Binary/Character

This instruction converts between binary (bit) strings and character (byte) strings containing X'F0' and X'F1'. The source field for the conversion is the data specified by operand 2. The result of the LCONVERT operation is placed in operand 1. This instruction has been implemented to facilitate support for COBOL and is not considered useful for general purpose application programming.

Note: LCONVERT is an optional instruction, requiring that optional module P34 be specified by the OPTMOD configuration macro.

Name	Operation	Operand
[label]	LCONVERT	{ defld1 | (defrf1) | (reg1) | seg1,disp1 } , { defld2 | (defrf2) | (reg2) | seg2,disp2,len2 } , { TOBITS | TOBYTES }

operand 1
: Is a field that receives the converted data. The field must not be in Segment 14 and the operand length is implied by the operand-2 length and the operation performed. If TOBITS is specified, the implied length of operand 1 is eight times that of operand 2. If TOBYTES is specified, the implied length is 1/8 that of operand 2. The length allowed for either operand 1 or 2 is from 1 to 4095 bytes.

operand 2
: Is a field containing the binary or X'F0/F1' character string to be converted. The operand length is from 1 to 255 unless you specify register addressing, which allows a length ranging from 1 to 4095 bytes.

TOBYTES
: Specifies a conversion from a character string of X'F0' and/or X'F1's to binary bits. X'F0' becomes a zero bit; X'F1', a one bit. Every eight X'F1/F0' characters become a byte of binary data. Any unfilled bit positions of the last byte are padded with zero bits. Conversion operates from left to right, and any operand-2 character bytes other than X'F0' or X'F1' cause unpredictable results.

TOBITS
: Specifies a conversion from a binary string to a character string of X'F0' and/or X'F1's. Each binary 1 bit becomes X'F1', and each binary 0 becomes X'F0'. Conversion operates from left to right.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, 03, 09, or 27 can be set.

Programming Notes: LCONVERT could be used to convert a flag byte used in an assembler application to an eight-byte character string that a COBOL application can examine. For example, using the LCONVERT instruction with the TOBITS operand:

INPUT	LCONVERT output
(operand 2)	(operand 1)
X'24'	X'F0F0F1F0F0F1F0F0'

LDDI--Load Data Immediate

LDDI loads a halfword of immediate data into the low-order 2 bytes of a register. The high-order 4 bytes are set to 0.

Name	Operation	Operand
[label]	LDDI	reg1,immdata2

operand 1
Is a register into which the immediate data is loaded.

operand 2
Is a 1- or 2-byte immediate data operand which becomes the halfword of immediate data. If a 1-byte numeric operand is used, it will be right-justified in the halfword of immediate data and the left byte will be set to 0. If a 1-byte character operand is used, it will be left-justified in the halfword immediate data, and the right byte will be set to a character blank.

Condition Code: The condition code is not changed.

Program Checks: None are set.

LDFLD--Load Field

LDFLD loads a field into a register. If the field is less than 6 bytes long, it is treated as a signed binary number, and the leftmost bit is propagated to the left in the register. If the field is more than 6 bytes long, truncation occurs on the left, and the condition code is set.

Name	Operation	Operand
[label]	LDFLD	reg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Is a register into which the field is to be loaded.

operand 2
Defines the field to be loaded. The length of the field is from 0 to 4095; operands greater than 15 bytes long must be selected using register addressing. If 0 is specified, the register is loaded with binary zeros.

Condition Codes: One or more of the following are set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0, or the length was specified as 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, 03, or 27 can be set.

LDFLDC--Load Field Character

LDFLDC loads the signed binary equivalent of a decimal EBCDIC number into a register. Because the largest field that can be loaded is 15 digits, the range of values that can be loaded is from −140737488355328 to 140737488355327. A negative number is stored in twos complement form. If the field to be loaded contains other than the digits 0 through 9 or a leading minus sign (hex 60), the results of the conversion to binary are unpredictable.

Name	Operation	Operand
[label]	LDFLDC	reg1, { defcon2 / defld2 / (defrf2) / (reg2) / seg2,disp2,len2 }

operand 1
Is a register into which the field is to be loaded.

operand 2
Is a field to be loaded. The length of the field is from 0 to 4095; operands greater than 15 bytes long must be selected using register addressing. If 0 is specified, a condition code of hex 01 is set, and the register is loaded with zeros.

Condition Codes: One or more of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0, or the length was specified as 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, 03, or 27 can be set.

LDFLDL--Load Field Logical

LDFLDL loads a field into a register. If the field is less than 6 bytes long it is treated as a 6-byte field by propagating a zero bit to the left in the register. If the field is more than 6 bytes long, truncation occurs on the left, and the condition code is set.

Name	Operation	Operand
[label]	LDFLDL	reg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

```
                           ( defcon2         )
                           | defld2          |
[label] LDFLDL    reg1,   <  (defrf2)         >
                           | (reg2)          |
                           ( seg2,disp2,len2 )
```

operand 1
Is a register into which the field is to be loaded.

operand 2
Is a field to be loaded. The length of the field is from 0 to 4095; operands greater than 15 bytes long must be selected using register addressing. If 0 is specified, the register is loaded with binary zeros.

Condition Codes: One or more of the following are set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0, or the length was specified as 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, 03, or 27 can be set.

Programming Note: The source field must not overlap the register field (registers are in Segment 0).

LDFP--Load Primary Field Pointer

LDFP loads the binary form of the PFP of a segment into the rightmost 2 bytes of a register. The leftmost 4 bytes are set to 0.

Name	Operation	Operand
[label]	LDFP	reg1,seg2

operand 1
Is the register that is to receive the PFP.

operand 2
Is the segment number (0-15) whose PFP is to be loaded.

Condition Codes: The code is not changed.

Program Checks (hex): 01 can be set.

LDLN--Load Field Length Indicator

LDLN loads the binary value of the field length indicator (FLI) of a segment into the rightmost 2 bytes of a register. The leftmost 4 bytes of the register are set to 0.

Name	Operation	Operand
[label]	LDLN	reg1,seg2

operand 1
Is the register that is to receive the FLI.

operand 2
Is the segment number (0-15) whose FLI is to be loaded.

Condition Codes: The code is not changed.

Program Checks (hex): 01 can be set.

LDRA--Load Register Address

This instruction creates a register address in operand 1. A register address has the following format:

Bits 00-07: (Set to zero)
08-11: Segment space ID
12-15: Segment
16-31: Length
32-47: Displacement

If operand 2 is a segment-displacement operand (that is; 'defcon2', 'defld2', or 'seg2,disp2,len2') then the register address loaded into operand 1 will contain: the current segment space ID and the segment number, length, and displacement associated with the operand.

If operand 2 is an unmodified register address (that is, '(reg2)') then the address will be loaded, as is, in operand 1.

If operand 2 is a modified register address (that is, '(defrf)') then the register address loaded into operand 1 will contain:

- the segment space ID from operand 2
- the segment number from operand 2
- the length from the referenced 'defrf'
- the sum of the displacement of the referenced defrf and the displacement in the register associated with the defrf.

Name	Operation	Operand
[label]	LDRA	reg1, { defcon2 / defld2 / (defrf2) / (reg2) / seg2,disp2,len2 }

operand 1
Is the register where the register address is created.

operand 2
Specifies the field or field label whose address becomes the register address in operand 1.

Condition Codes: The code is not changed.

Program Checks (hex): 01 and 27 can be set.

LDREG--Load Register

LDREG replaces the contents of one register with the contents of another register. The contents of reg2 are not changed. Both operands may specify the same register to test its contents.

Name	Operation	Operand
[label]	LDREG	reg1,reg2

operand 1
Is the register to be loaded.

operand 2
Is the register whose contents are to be loaded into the first register specified.

Condition Codes: One of the following is set:

Hex Code	**Possible Mnemonic**	**Explanation**
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.

Program Checks: None are set.

LDSECT--DSECT Definition (BEGIN)

The LDSECT instruction starts the definition of a sequence of level (Ln) instructions. Each level instruction associates a label with a corresponding length and displacement. The LDSECT instruction options determine whether the above label (field) is defined within a segment or as an offset from a register.

Name	Operation	Operand
label	LDSECT	{ SDL= {seg|defld} / BASE= { reg|*} } [,DISP=number]

SDL
Either a segment number or the label of a defld instruction.

BASE
A register number that is used, assumed to contain a register address locating the fields of the DSECT.

DISP
Either an '*' or a number that specifies the starting displacement of the fields of the DSECT. An '*' indicates the fields should start after the previous DEFLD or DEFRF specification for the specified segment.

Note: The SDL and BASE keywords are mutually exclusive.

If no keywords are specified then defaults are set as follows:

1. Level instructions define fields within a segment.

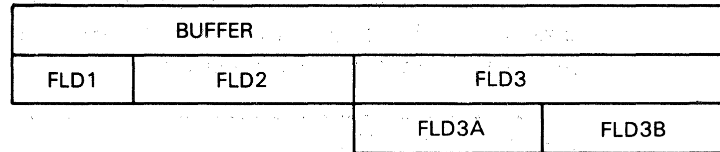
2. The starting displacement and segment number are defaulted to the last DEFLD instruction's values encountered prior to the LDSECT instruction.

You can use the SDL and DISP keywords in conjunction as follows:

1. SDL specifies that the level instructions define fields within a segment and the SDL operand specifies the segment number. If SDL specifies the label of a DEFLD instruction then the field starts within the segment and at the displacement specified on the DEFLD instruction.

2. DISP specifies the starting displacement within the segment. The DISP operand overrides the DEFLD displacement of SDL. If SDL specifies a DEFLD label and DISP specifies an '*', then only the segment value of the SDL specification is used.

Programming Notes: The following are examples of how Level definition instructions may be used to describe a data area.



In this example assume that the data area is to be located by register 5.

```
BUFF      LDSECT  BASE=5
          L1      BUFFER
            L3   FLD1,10
            L3   FLD2,10
            L3   FLD3,40
            L3   FLD3AB,REDEF=FLD3
              L5 FLD3A,20
              L5 FLD3B,20
          LEND   PRINT=ON
```

```
BUFFER    DEFRF    5,0,60
FLD1      DEFRF    5,(BUFFER),10
FLD2      DEFRF    5,,10
FLD3      DEFRF    5,,40
FLD3AB    DEFRF    5,(FLD3),40
FLD3A     DEFRF    5,(FLD3AB),20
FLD3B     DEFRF    5,,20
```

The following is another example of a DSECT definition:

```
BUFF      LDSECT  SDL=3,DISP=0
          L1      BUFFER
            L3   FLD1,10
            L3   FLD2,10
            L3   FLD3,40
            L3   FLD3AB,REDEF=FLD3
              L5 FLD3A,20
              L5 FLD3B,20
          LEND   PRINT=ON
```

The following instructions are created from the coding above:

```
BUFFER    DEFLD    3,0,60
FLD1      DEFLD    3,BUFFER,10
FLD2      DEFLD    3,,10
FLD3      DEFLD    3,,40
FLD3AB    DEFLD    3,FLD3,40
FLD3A     DEFLD    3,FLD3AB,20
FLD3B     DEFLD    3,,20
```

LDSEG--Load Segment

LDSEG places the contents of a Segment 1-header addressed field into a register. If the field is longer than 6 bytes, truncation occurs on the left; if the field is less than 6 bytes, it is treated as a signed binary number, and the leftmost bit is propagated to the left in the register.

Name	Operation	Operand
[label]	LDSEG	reg1,seg2

operand 1
Is a register that is to contain the result of the operation.

operand 2
Is a field in the specified Segment 1 to be loaded into the register. The location of the field within the Segment 1 is determined by the PFP, and its length by the FLI. If the field length is 0, the register is loaded with binary zeros.

Condition Codes: One or more of the following are set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0 or the field length was specified as 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	TR	Truncation occurred.

Program Checks (hex): 01 and 03 can be set.

LDSEGC--Load Segment Character

LDSEGC loads the binary equivalent of a Segment 1-header addressed field into a register. The range of values that can be loaded is from −140737488355328 to +140737488355327. A negative number is loaded in twos complement form. If the field contains other than the digits 0 through 9 or a leading minus sign (hex 60), the results of the conversion to binary are unpredictable.

Name	Operation	Operand
[label]	LDSEGC	reg1,seg2

operand 1
Is the register into which the field is to be loaded.

operand 2
Is a field in the specified Segment 1 to be loaded into the register. The location of the field within the Segment 1 is determined by the primary field pointer, and its length by the field length indicator. The field length can be from 0 to 4095 bytes. If the field length is 0, the register is loaded with binary zeros.

Condition Codes: One or more of the following are set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0 or the length is specified as 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.

Program Checks (hex): 01 or 03 can be set.

LDSEGLN--Load Segment Length

LDSEGLN loads the length of the Segment 1 (in binary) into a register.

Name	Operation	Operand
[label]	LDSEGLN	reg1,seg2

operand 1
Is the register that is to contain the Segment 1 length.

operand 2
Is the Segment 1 number whose length is to be loaded.

Condition Codes: The code is not changed.

Program Checks (hex): 01 can be set.

LDSFP--Load Secondary Field Pointer

LDSFP loads the secondary field pointer (SFP) in binary into a register. The SFP is loaded into the rightmost 2 bytes of a register and the leftmost 4 bytes are set to zero.

Name	Operation	Operand
[label]	LDSFP	reg1,seg2

operand 1
Is the register that is to contain the SFP.

operand 2
Is the Segment 1 number whose SFP is to be loaded.

Condition Codes: The code is not changed.

Program Checks (hex): 01 can be set.

LEJECT--Eject to a New Page

The LEJECT instruction is used to separate pages of print in the output listing of your program. When issued, the LEJECT instruction will skip to the first printable line of a new page. The standard EJECT instruction of the DOS/VS or OS/VS assembler may be included, but will be ignored.

Name	Operation	Operand
[label]	LEJECT	

LEND--DSECT Definition (End)

The LEND instruction is used to indicate the end of a set of level (Ln) instructions. It causes the generation of either DEFLD's or DEFRF's that represent the preceding level instructions.

Name	Operation	Operand
	LEND	[PRINT= {OFF|ON}]

PRINT
Is an optional operand that specifies whether the DEFLD or DEFRF instructions are to be listed (ON) or not listed (OFF). OFF is the default.

LEXEC--Execute

The LEXEC instruction specifies the label of an instruction and a register. The labeled instruction is ORed with all or part of the data in the register. The resulting subject instruction is then executed. Neither the labeled instruction nor the register is altered by the ORing.

The execute instruction performs the following sequence of steps to create the subject instruction:

1. The labeled instruction is placed, left-justified, into an 8-byte buffer of zeros. The length of the labeled instruction is determined by its operation code. (The 8-byte buffer is internal to the controller.)

2. If register number 0 is specified by the execute instruction, no ORing takes place and the labeled instruction is the subject instruction.

   If the register number is not 0, then the rightmost 2, 4, or 6 bytes of the register are ORed with the leftmost 2, 4, or 6 bytes of the buffer, respectively. The number of bytes ORed is determined by an operand of the execute instruction.

3. The subject instruction is now left-justified in the buffer. Notice that the length of the subject instruction is a function of the operation code of the subject instruction, and the length may be 2, 4, 6, or 8 bytes.

The subject instruction may be any valid executable instruction except a jump, branch, or another LEXEC.

LEXEC supports a WRTI subject instruction only if SPLIT=N is coded or defaulted on the APOPT instruction and the immediate data is in the addressed instruction. (WRTI is in *4700 Programming Library, Volume 4*.) Control continues with the instruction following the LEXEC after the subject instruction completes, unless the subject instruction is an LEXIT, LSEEK (that branches), or instructions that modify SMSUIC.

The ORing of the labeled instruction with the register permits control of, for example, the operation code or Segment 1 number at execution time.

Name	Operation	Operand
[label]	LEXEC	reg1,addr [,len|0]

reg

Is the register to be used when forming the subject instruction. If you specify register 0, the labeled instruction is used as the subject instruction; that is, no ORing takes place.

addr

Is the label of the instruction (within Segment 14) which is to be ORed to reg1 when forming the subject instruction. The value of "addr" must be even and may be an external symbol when RELOC=Y is specified in the APOPT instruction.

len

For a register other than 0, len determines the number of bytes to be ORed. Valid values for *len* are 0, 2, 4, and 6. The default value is zero. A value of 0 indicates that the length of the labeled instruction is the number of bytes to OR. Values of 2, 4, and 6 indicate that 2, 4, and 6 bytes are to be ORed, respectively. The two low-order instruction bytes cannot be changed for 8-byte instructions.

len must be

- an unsigned decimal integer
- the label of an EQUATE instruction that is associated with a decimal integer.

Condition Code: The condition code can be set by the subject instruction.

Program Checks (hex): Program checks can be generated by the LEXEC or the subject instruction; in either case, it is the address of the LEXEC instruction that is flagged as having caused the program check. The LEXEC instruction can cause the following program checks:

03 = field length error (*len* not 0, 2, 4, or 6)

0B = instruction address error

11 = subject instruction cannot be performed by LEXEC

Programming Notes: The following example illustrates a use of LEXEC to create and execute the subject instruction LDSEGLN R03, S04.

```
GETLN0  LDSEGLN  R03,0       --  addressed instruction
          •
          •
          •
        LDDI     5,X'04'     --  load register 5 with ORing value
GETLN4  LEXEC    5,GETLN0    --  create and execute subject instruction;
          •                      addressed instruction      ... 1D30
          •                      low-order 2 bytes of reg 5 ... 0004
          •                      executed subject instruction .. 1D34
        nsi                      control returns to nsi
```

LEXIT--End of Processing

LEXIT signals the end of current processing for a station (usually, the end of a transaction). It waits until all devices assigned to the station are not busy and until all activity with the communication link is in a quiesced state. If any outstanding responses are required from a previous read operation with the host processor, LEXIT causes them to be sent. When the instruction is performed, the following fields in Segment 1 are set to 0:

Relative instruction counter (SMSUIC)
Pause Instruction Counter (SMSPCT)
Indicator byte (SMSIND)
Terminal group unit (SMSTGU)

The instruction loop threshold count (SMSLTC) is set to 1 and the flag bit in SMSFG2 indicating program check routine in control is set off.

The station ID is also restored (SMSSID in Segment 1) to ensure that the correct ID exists, and SMSLSE is set to the value in SMSLSB.

After execution of LEXIT, the applicable station stops processing until an asynchronous entry point is activated for the station. The station relinquishes control even if an asynchronous input is pending at the time of the LEXIT execution.

Note: When LREAD NOWAIT for a keyboard/display precedes the LEXIT, the operator must complete the input operation before the LEXIT is performed. See the *4700 Programming Library, Volume 4.*

A work terminal sharing a terminal with another station can relinquish control of that terminal before issuing LEXIT by using an ASSIGN instruction. (See the description of "ASSIGN" in Volume 4.)

Name	Operation	Operand
[label]	LEXIT	

Condition Codes: The code is not changed.

Program Checks: None are set.

LHRT--Load High-Resolution Counter

This instruction loads the high-resolution counter (HRC) into a six-byte field defined by the instruction. The low-order byte of the field contains 1/256 seconds, and the high-order five bytes contain the seconds. The time measured by the HRC is the total time elapsed since the last controller load (IPL).

Name	Operation	Operand
[label]	LHRT	{ defld2 / (defrf2) / (reg2) / seg2,disp2 }

operand 2

Is a field to contain the timer value read from the HRC. The field length is six bytes, and any specified length is ignored. The field must not be in Segment 14.

Condition Codes: The condition code is not changed.

Program Checks (hex): 01, 02, or 27 can be set.

Programming Notes: The following example uses LHRT to compute the time to perform operation X:

```
BEGIN   DEFLD    3,,6        6 BYTE SAVE AREA IN SEGMENT 3
END     DEFLD    3,,6
        •
        •
        LHRT     BEGIN       SAVE BEGIN COUNTER VALUE
        •
        • (Operation X)
        •
        LHRT     END         GET CURRENT COUNTER VALUE
        LDFLD    2,END
        SUBFLD   2,BEGIN     REG2 = INTERVAL
```

The error factor of the interval measured is always less than the resolution of the counter. For example, the computed interval would be almost 1/256 second too large if the first LHRT was executed just before the counter increased, and the second LHRT was executed just after the counter increased.

LIFOFF--If Off Then Branch

LIFOFF changes the sequence of program execution by causing a branch when a specified bit is off. If the specified bit is on, no branching occurs. The ELSE operand can be coded to set the specified bit off if the branch is not taken and the next sequential instruction is executed.

Name	Operation	Operand
[label]	LIFOFF	{ defld2 | (defrf2) | (reg2) | seg2,disp2 } ,bitnum,branch address [,ELSE=SETOFF]

operand 2

Is a field containing the bit to be tested. The length associated with this fixed field is ignored; the field is assumed to be 2 bytes long. If ELSE=SETOFF is coded, the Segment 1 number associated with the field may not be 14.

bitnum

Is the bit number (0-15), within the 2-byte field, to be tested. Bit number 0 is the leftmost bit of the first byte. One of the following must be used for bitnum:

1. An unsigned decimal integer (0 to 15) representing the bit to be tested.

2. A 1- or 2-byte hexadecimal value which, after translation, represents the bit to be tested. The following illustrates valid hexadecimal values and the resulting translation:

Hex	Bit to be Tested
X'80'	0
X'40'	1
X'20'	2
•	
•	
X'02'	6
X'01'	7
X'8000'	0
X'4000'	1
•	
•	
X'0002'	14
X'0001'	15

3. A label of an EQUATE instruction associated with a decimal or a hexadecimal value (as described above) that identifies the bit to be tested.

branch address

Is the label of the instruction to be executed if the bit tested is off.

ELSE=SETOFF

If used, will set the tested bit off if the branch is not taken. If it is omitted, no change is made to the bit tested.

Condition Code: The condition code is not changed.

Program Checks (hex): 01, 02, or 0B can be set.

LIFON--If On Then Branch

LIFON changes the sequence of program execution by causing a branch when a specified bit is on. If the specified bit is off no branching occurs. The ELSE operand can be coded to set the specified bit on if the branch is not taken and the next sequential instruction is executed.

Name	Operation	Operand

```
                    ⎧ defld2      ⎫
[label] LIFON       ⎨ (defrf2)    ⎬ ,bitnum,branch address
                    ⎪ (reg2)      ⎪
                    ⎩ seg2,disp2  ⎭

                    [,ELSE=SETON]
```

operand 2

Is a field containing the bit to be tested. The length associated with this fixed field is ignored; the field is assumed to be 2 bytes long. If ELSE=SETON is coded, the segment number associated with the field may not be 14.

bitnum

Is the bit number (0-15), within the 2-byte field, to be tested. Bit number 0 is the leftmost bit of the first byte. One of the following must be used for bitnum:

1. An unsigned decimal integer (0 to 15) representing the bit to be tested.

2. A 1- or 2-byte hexadecimal value which, after translation, represents the bit to be tested. The following illustrates valid hexadecimal values and the resulting translation:

Hex	Bit to be Tested
X'80'	0
X'40'	1
X'20'	2
•	
•	
X'02'	6
X'01'	7
X'8000'	0
X'4000'	1
•	
•	
X'0002'	14
X'0001'	15

3. A label of an EQUATE instruction associated with a decimal or a hexadecimal value (as described above) that identifies the bit to be tested.

branch address

Is the label of the instruction to be executed if the bit tested is on.

ELSE=SETON

If used, will set the tested bit on if the branch is not taken. If it is omitted, no change is made to the bit tested.

Condition Code: The condition code is not changed.

Program Checks (hex): 01, 02, or 0B can be set.

Programming Notes: In the following example, a work station has established a 'FLAGS' field in one of its private storage Segment 1s. FLAGS is being used as a bit switch field by the work station and switch 4 is being used to indicate whether process x and y or just process y must be executed. When the switch is on, process y is to be executed; when the switch is off, process x and y are to be executed. Switch 4 is assumed to be set by another routine.

```
FLAGS   DEFLD    2,,2                (1)
SW4     EQUATE   4                   (2)
START     •
          •
          •
        LIFON    FLAGS,SW4,SKIP      (3)
        (nsi)    (begin process x)
          •
          •
          •
          •      (end process x)
SKIP      •      (begin process y)
          •
          •
          •
```

In this example:

statement (1) defines the 2-byte field of bit switches; statement (2) is the equate for bit switch 4 (X'0800'); and statement (3) controls program flow -- if the switch is on, a branch to SKIP is taken, and only process y is executed; if the switch is off, both process x and process y are executed.

LLOAD--Load an Overlay Section into Main Storage

LLOAD loads an application overlay into main storage. If a register other than 0 is specified, the location of the first instruction in the overlay section is returned in the specified register. This location can be used to branch to an overlay section that was assembled independently of the root section.

Two different types of LLOAD operations can be performed, depending on whether or not the parameter PARM=EXP is specified. A 'normal' LLOAD operation is performed if PARM=EXP is not coded. In this case, the LLOAD instruction loads the specified overlay into main storage only if it is not already loaded. The location where the overlay is loaded is derived from the Resident Overlay Directory (ROD).

An 'expanded' LLOAD operation is performed if the parameter PARM=EXP is coded in the LLOAD instruction. The 'expanded' LLOAD operation loads overlays into main storage at locations different from those defined in the ROD; in this case, the load address(es) where the overlay is to be loaded must be specified explicitly by you in the expanded LLOAD parameter list.

Name	Operation	Operand
[label]	LLOAD	[reg1], { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2 } [,PARM=EXP]

operand 1
Is the register that is to contain the location of the overlay section's first instruction after this LLOAD is executed. LLOAD sets operand 1 to zero if a split overlay is loaded with no instruction section, or if an expanded LLOAD loads a nonsplit overlay into a segment other than 14. If operand 1 is omitted or specified as 0, the location of the overlay section's first instruction is not returned, but the execution of the LLOAD still takes place.

operand 2
Defines the name of the overlay when PARM=EXP is not coded, or defines the expanded LLOAD parameter list when PARM=EXP is coded. Any length is ignored. If PARM=EXP is not coded, the first 8 bytes are assumed to be the name of the overlay. If PARM=EXP is coded, the first 14 bytes are assumed to be the expanded LLOAD parameter list.

PARM=EXP
If coded, will perform an expanded LLOAD operation. The load address(es) for the overlay must be specified in the expanded LLOAD parameter list.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	The instruction was executed successfully.
02	ST	Status is stored. The status code is contained in SMSDST. This instruction may produce status that is defined in the *IBM 4700 Controller Programming Library Volume 2: Disk and Diskette Programming.*

Program Checks (hex): 01, 02, 03, 06, 08, or 27 can be set. If PARM=EXP is coded, 11 can also be set.

Programming Notes: LLOAD loads the application overlay into Segment 14 as defined by the STOVLY instructions in the calling routine and the OVLYSEC instruction at the beginning of the overlay.

Operation of the LLOAD instruction includes data transfer because application overlays reside on a disk or diskette. Unsuccessful operations set a condition code to indicate that status has been stored. (Application programs, including overlays, are placed on a disk if CPYAP=Y and DSK=A were coded in the FILES configuration macro.)

An expanded LLOAD operation loads overlays into user storage under control of the expanded LLOAD parameter list. When you code PARM=EXP in the LLOAD, overlays can be loaded into various locations in storage. You can specify load addresses in the expanded parameter list or you can allow them to default to the normal load addresses. Addresses in the parameter list are used if you set the appropriate flags.

Expanded LLOAD can be used when more than one station share a common application program containing a number of overlay sections, and the overlay sections are all defined to use the same storage space.

The normal LLOAD operation does not load an overlay into more than one location. One station could load an overlay while another station required the same space to load another overlay from the same program. The latter station would have to wait until the first station completed operation and released the space. You can avoid this situation by using the expanded LLOAD capability. You must define a set of storage areas where any overlay can be loaded. Your program must manage these storage areas by keeping track of whether they are 'occupied' or 'free'.

The definition of the LLOAD parameter list is provided by the COPY DEFELP instruction. (See Appendix B for details.)

The following describe the parameter list entries for expanded LLOAD:

Field	Contents
ELPOVN Bytes 0-7	The 8-byte name of an existing overlay. This name is used by LLOAD to locate the overlay.
ELPFLG Byte 8	This field contains the flags that indicate which parameter list entries, if any, should be used to determine the relative load address or addresses for the overlay. If neither flag is set, a program check (hex 17) will occur. Flag bit masks: ELPCSF This flag causes LLOAD to derive load addresses This flag causes LLOAD to derive load addresses for the overlay from the ELPSEG and ELPCLA fields. ELPISF This flag causes the load address to be taken from the ELPILA field.
ELPSEG Byte 9	This field specifies the segment (0 - 15) where the overlay will be loaded. It is used only if the appropriate flag is set.
ELPCLA Bytes 10,11	This field specifies the relative address within the segment defined by ELPSEG, where the overlay will be loaded. This value must be an even number. If ELPSEG specifies Segment 14 then this value must *not be less than* 48 (hex 30). This field is used only if the appropriate flag is set.
ELPILA Bytes 12,13	This field specifies the relative address within the application program for the overlay. This value must be even; and must be *greater than or equal to* 48 (hex 30). This field is used only if the appropriate flag is set.

If the overlay section is to be loaded at a location other than that defined in the ROD, the use of branching instructions that refer to locations within the overlay causes incorrect operations to occur. Branching instructions cannot be used to transfer from outside an overlay (such as from a root section) to a point within an overlay.

In the same sense, do not code the OVLYEP parameter of the OVLYSEC instruction if the overlay is to be loaded in a location other than that defined in the ROD. Coding OVLYEP creates a branch to the entry point label. The entry point of such an overlay should always be the instruction following the 2-byte header in the overlay, or the entry point must be calculated before the branch to the entry point occurs.

The same restriction applies when referencing DEFLD or DEFCON instructions that refer to labels within the overlay. All references should instead first calculate the current overlay load address, and then add the offset to the desired data fields within the overlay.

LMERGE--Merge Blocks of Records

LMERGE collates sequenced data items from two separate input blocks into a third output block as specified by the parameters contained in a list. After the merging is completed, execution continues with the next sequential instruction (unless a program check occurs).

The instruction points to a parameter list (see the DEFMER copy file) consisting of block pointers, data item descriptions, and other variables. Two input blocks of fixed-length data items are merged into an output block. Each data item contains a collating key at a fixed displacement in the item. The data items in each block occupy contiguous storage space.

Note: This is an optional instruction that requires optional module P5C be specified in the OPTMOD configuration instruction.

In general, LMERGE attempts to place one of the two current items from the input blocks into the next available portion of the output block. LMERGE tries to place the current item with the smaller (larger) key into the output block for an ascending (descending) merge. The current item from the input block, which is tried first, is called the primary item; the other input block's current item is called the secondary item.

The following definitions will help clarify LMERGE.

Current Item of an Input Block- is the data item being considered for placement into an output block.

Current Item of an Output Block - is the next available position in the output block.

Data Set Swap Flag - is a bit switch in the return byte of the LMERGE parameter list. It will be switched on when a sequence check occurs on a merge unit boundary. The flag can be used in a sort application that is data driven.

Empty Input Block - is an input block that LMERGE has marked as empty (by setting a bit in the return flag byte of the LMERGE parameter list) because the last physical item has just been copied from the input to the output block.

Fragmented Key - A key of a data item is said to be fragmented if its contents (from most significant to least significant) are not sequential from left to right.

Merge Unit - is the size, in bytes, of the smallest string of ordered items that are to be read from or written to a disk or diskette.

Merge Unit Boundary - is used to test for a sequence check that will occur if:

- Both the primary and secondary items of the input blocks are out of sequence with respect to the previous data item of the output block.

- The current output pointer minus the displacement to the begin pointer is a multiple of the Merge Unit.

Null Input Block - is an input block that the application program has marked as null (by setting a bit in the input flag byte of the parameter list). LMERGE will not copy items from a null input block.

Previous Item - is the item most recently copied to the output block. The displacement to the previous item plus the item length equals the displacement to the current item of the output block.

Primary Item - is the item with the smaller key for an ascending merge (or larger key for a descending merge) of the 2 input blocks. If the keys are equal then the primary item is the current item from block 1. (If one input block is null then the primary item is the current item from the non-null block.)

Secondary Item- is the item with the larger key for an ascending merge (or smaller key for a descending merge) of the 2 input blocks. If the keys are equal then the secondary item is the current item from block 2. (If one input block is null then the secondary item is the current item from the non-null block.)

Sequence Check - occurs during operation of LMERGE when both current items of the input blocks are out of sequence with the previous item of the output block. If one of the input blocks is a null block, a sequence check occurs when the active input block current item is out of sequence with the previous item of the output block.

LMERGE operates according to the following rules:

When invoking LMERGE or after the output block contents have been written to a data set because of a sequence check, the key of the physically last data item in the output block should be set to hex 00s (ascending merge) or hex FFs (descending merge). This will avoid an unwanted sequence check.

If the current pointer of any block equals the end pointer of the block, LMERGE begins by setting the current pointer equal to the begin pointer of the block. Therefore, having the current pointer equal to the begin pointer and having the current pointer equal to the end pointer are equivalent.

If the input flag byte indicates that both input blocks are null, LMERGE ends with the both-input-blocks-null bit on in the return flag byte.

If the input flag byte indicates that one input block is null, a one-way merge is performed, using the other block as the sole input source. The input-block-empty bit in the return flag byte is not set on for the null block.

If the key of the previous item of the output block is in sequence with the key of the primary item, the primary item is copied into the current position of the output block. The current pointer of the block that contained the primary item is increased by the item length, and, if its new value equals the block's end pointer, the appropriate input-block-empty bit of the return flag byte is switched on.

If the key of the previous item of the output block is not in sequence with the key of the primary item, and is in sequence with the secondary item's key, the secondary item is copied into the current position of the output block. The current pointer of the block that contained the secondary item is increased by the item length, and, if its new value equals the block's end pointer, the appropriate input-block-empty bit of the return flag byte is set.

After a data item is copied to the output block, the previous item becomes the item just moved, and the output block's current pointer is increased by the item length. If this new current pointer equals the end pointer, the output-block-full bit of the return flag byte is switched on.

If any bit or bits of the return flag byte have been switched on, execution of the next sequential instruction begins; otherwise, an attempt is made to copy another data item to the output block.

The merge output block should always be written out to a disk or diskette data set (unless the current pointer equals the begin pointer) when the return flag byte requests processing (output block full, sequence check on a merge unit boundary, or both input blocks null) before another LMERGE is executed.

If the key of the previous item of the output block is out of sequence with both the primary and secondary item's keys:

1. Neither input item is copied into the output block.

2. The sequence check condition code is set.

3. If the merge unit is not zero and the current pointer minus the begin pointer of the output block is a multiple of the merge unit, the sequence-check-on-a-merge-unit-boundary of the return flag is set. Otherwise, the sequence-check-not-on-a-merge-unit-boundary is set.

4. LMERGE processing ends and program operation continues with the next sequential instruction.

LMERGE Parameter List: The parameter list has the following format:

Displ	Size	Parameter
0	1	Input flag byte. The following bits are set/reset by the application program. **Bit** **Meaning** 0 =0 indicates ascending keys =1 indicates descending keys 1-5 Reserved 6 =1 indicates input block 1 null 7 =1 indicates input block 2 null
1	1	Return Flag Byte The following bits are set/reset by each operation of LMERGE: **Bit** **Meaning** 0 Output block full (Note: The current pointer of the output block equals its end pointer.) 1 Input block 1 empty (Note: The current pointer of input block 1 equals its end pointer.) 2 Input block 2 empty (Note: The current pointer of input block 2 equals its end pointer.) 3 A sequence check occurred, and the current pointer for the output block was not on a merge unit boundary. 4 Data-set swap flag: A sequence check occurred, and the current pointer for the output block was on a merge unit boundary. 5 Both input blocks were null input blocks (that is; both null-block of the input flag byte are set). 6-7 Reserved.

Displ	Size	Parameter
2	2	Displacement to the beginning of the first input block within its segment (begin pointer).
4	2	Displacement to the end of the first input block within its segment (that is, the byte following the last item) (end pointer).
6	2	Displacement to the beginning of the second input block within its segment (begin pointer).
8	2	Displacement to the end of the second input block within its segment (that is, the byte following the last item) (end pointer).
10	2	Length of each data item.
12	2	Displacement to the collating key from the start of an item.
14	1	Length of the collating key.
15	1	The number of the segment containing the first input block.
16	1	The number of the segment containing the second input block.
17	1	The number of the segment containing the output block. This may not be 14.
18	2	Displacement to the beginning of the output block within its segment (begin pointer).
20	2	Displacement to the end of the output block within its segment (that is; the byte following the last item) (end pointer).
22	2	Displacement to the current item in the first input block (initially set to the beginning of the block and subsequently modified by the controller during LMERGE processing). This current item is the next entry from input block 1 to be considered for placement into the output block (current pointer).
24	2	Displacement to the current item in the second input block (initially set to the beginning of the block and subsequently modified by the controller during LMERGE processing). This current item is the next entry from input block 2 to be considered for placement into the output block (current pointer).
26	2	Displacement to the current item in the output block (initially set equal to the beginning of the block and subsequently modified by the controller during LMERGE processing). The current item in the output block is the next position available to receive a data item (current pointer).
28	2	Merge unit. This is the size in bytes of the smallest string of ordered items that are input from, or output to, a data set while merging. The value of the merge unit determines which sequence check bit of the return flag byte is switched on if a sequence check occurs. If the merge unit is set to zero by the application program, and if a sequence check does occur, then the sequence-check-not-on-a-merge-unit-boundary bit is switched on. If the merge unit is not zero, then the merge output block should be a multiple of the merge unit.

Name	Operation	Operand
[label]	LMERGE	{ seg2 / defld2 / (defrf2) / (reg2) / seg2,disp2 }

operand 2

is a field containing the parameter list. The field cannot be in Segment 14. If you specify seg2 only, the displacement of the list within the segment is determined from the primary field pointer (PFP). Any field length is ignored.

When using register addressing to locate a parameter list it can be in noncurrent segment space. However, if the parameter list contains addresses (segments, displacements) of other storage areas then these storage areas must be in the current segment space.

Condition Codes: At completion of LMERGE execution, one of the following condition codes is set:

Hex Code	Explanation
01	Sequence check did not occur.
02	Sequence check did occur.

Below is a table of all possible combinations of condition codes (CC) and bits returned in the return flag byte of the parameter list:

CC:	Flag:	Description:
01	80	Output block full.
01	40	Input block 1 empty.
01	C0	Input block 1 empty and output block full.
01	20	Input block 2 empty.
01	A0	Input block 2 empty and output block full.
01	04	Both input blocks were null input blocks.
02	10	Sequence check, not on merge unit boundary.
02	08	Sequence check, on merge unit boundary.

Program Checks (hex): 01, 02, 09, 11, or 27 can be set.

Program check 01 results from one of the following conditions:

- Parameter list is located in an invalid or undefined segment.
- Invalid or undefined segment referred to in the parameter list.

Program check 02 results from one of the following conditions:

- The parameter list is not completely contained within the given segment.
- Input block 1, input block 2, or output block is not completely contained within its segment.

5-211 to 5-258
missing from original
document

Field	Value	
IND	8	
INL	2	
OUD	6	
OUL	3	
LIC	X'4C'	
FNC	X'3F'	
LID	7	
LOD	4	
CNT	6	
TST	X'80'	-- indicates ending on translation break.

Data Translation Example 2: In this example, two translation tables are generated, each with four entries. The input code ranges are X'1B' to X'1E', and X'61' to X'64'.

> **Note:** Generating one table with a range of X'1B' to X'64' would result in a table with 74 entries, requiring more storage space.

In addition to the operations of example 1, translation control functions for case shift (up and down -- CASn and DS), transparent write (TW), and positioning (ADV) are illustrated in this example, as well as a demonstration of the handling of an input stream with exceptional data (that is, data that is not to be processed by LTRT). In this example, the exceptional data is 2 bytes long, and is immediately preceded by the input code of X'61'. Routine SPEC61 is designed to process this data. This example results in two executions of LTRT.

```
        BEGIN   ...,DSECT=Y,...

        DEFTRPS                        EQUATE 2
        COPY    DEFTRP                      Note 1
TRANS2  DEFLD   DEFTRPS,,4
           •
           •
INPUT   DEFLD   S3,1,14                     input area's location
OUTPUT  DEFLD   S4,1,12                     output area's location
           •
           •
PLIST   DEFCON  AL2(3),                INS=3
                AL2(INPUT-BQK$S3),     IND=1
                AL2(L'INPUT),          INL=14
                AL2(4),                OUS=4
                AL2(OUTPUT-BQK$S4),         OUD=1
                AL2(L'OUTPUT),         OUL=12
                X'80',                 MSK=X'80'
                AL1(2),                NTT=2
                AL1(1),                PTT=1
                AL1(1),                CTT=1
                XL10'00',              fill
                AL2(14),               TTS1=14
                YL2(TTBLY),            TTD1=addr of TTBLY in seg 14
                AL2(14),               translation table 2's segment
                YL2(TTBLZ)             translation table 2's addr
```

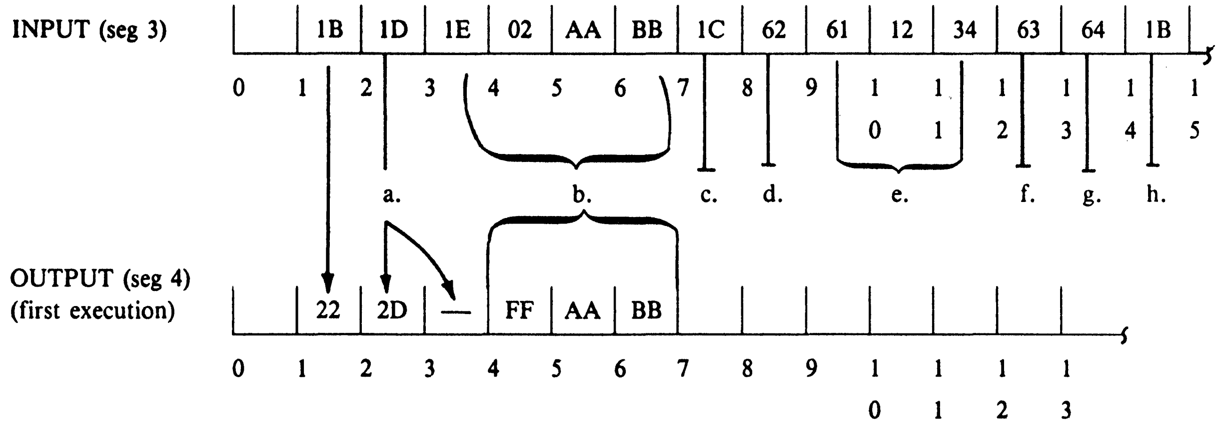
Notes:

1. Define LTRT parameter list fields in Segment 2 and define an additional 4-byte field (TRANS2) for the segment number and displacement of the second translation table.

```
                  •
                  •
         LTRTBEG                                    RANGE=(X'1B',X'1E',SKIP)
         LTRTENT                                    (X'1B',X'22'),
                  (X'1C',,CAS2),
                  (X'1D',X'2D',ADV),
                  (X'1E',X'FF',TW)
TTBLY    LTRTGEN                                    generate table 1
         LTRTBEG                                    RANGE=(X'61',X'64',X'00')
         LTRTENT                                    (X'61',,,X'C1',SPEC61,3),
                  (X'62',,,X'7F'),
                  (X'63',X'20')
                  (X'64',X'21',DS)
TTBLZ    LTRTGEN                                    generate table 2
                  •
                  •
         MVFXD    TRPPAR,PLIST                      Note 2
CONT     LTRT     TRPPAR
                  •
                  •
         LEXIT                                      Note 3
SPEC61            .
                  •
                  •
         BRAN     CONT
                  •
                  •
                  •
```

2. Initialize LTRT parameter list.
3. For the sample input shown following, LTRT ends processing of input code X'61' with a branch to SPEC61. When SPEC61 makes the branch to CONT, LTRT begins operation for a second time. Note that between the first and second operations of LTRT, the parameter list has not been reinitialized.

The translation process for the first operation of LTRT follows:



1. Translation definition and ADV translation control definition. Output, starting with the next input code, is spaced forward one position. Byte 3 of Segment 4 was not modified by this operation.

2. Translation definition and TW translation control definition. AA and BB are passed to the output area as is, although out of range. 02 is the length specification for the transparent write.

3. CAS2 translation control definition causes shift to translation table 2 (TTBLZ). In the parameter list, PTT is set to 1 and CTT is set to 2.

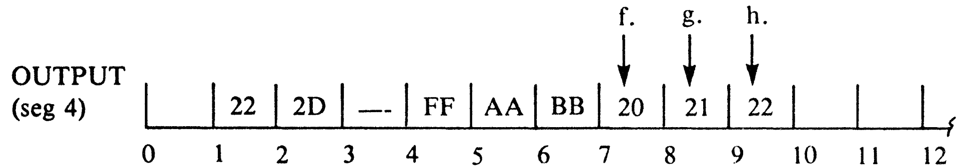
4. Program control definition; however, no bits in the mask specified match any bits in the parameter list mask. Translation continues.

5. Program control definition and translation control definition. LTRT operation ends with a condition code of 02 and status of translation break. Program flow branches to SPEC61 to process the exceptional data (1234) following input code 61. SPEC61 does not place any data in the output area. When SPEC61 branches back to CONT, LTRT operation resumes, starting with input code 63 because the translation control definition associated with 61 specified a displacement of three positions to the next input code to be processed. Because the parameter list is not reinitialized between operations, the input displacement value (IND) field will be set to 12 when the second LTRT operation starts.

The fields of the parameter list that are set or modified at the end of the first LTRT operation are shown below, along with the values resulting from this first operation of LTRT:

Field	Value
IND	12
INL	3
OUD	7
OUL	6
PTT	1
CTT	2
LIC	X'61'
FNC	X'00'
LID	9
LOD	7
CNT	6
TST	X'80' -- indicates ending on translation break

When the branch is made to CONT, LTRT begins operation for the second time. This time, for the sample input, translation will end with condition code 01 (input exhausted) and program operation will continue with the next sequential instruction. At the end of this second operation, the output area appears as follows:



6. Translate definition only.

7. Translation definition and DS translation control definition. Translation control shifts back to table 1 (TTBLY).

8. Translate definition only.

The parameter list fields that are modified or set by LTRT operation appear as follows after this second operation:

Field	Value
IND	15
INL	0
OUD	10
OUL	3
PTT	1
CTT	1
LIC	X'1B'
FNC	X'00'
LID	14
LOD	9
CNT	3 (for 2nd operation only)
TST	X'00' -- indicates translation was completed (all input data processed)

LTRTBEG--Translate Table Begin

LTRTBEG is used to define the range of input codes in the translation table. The table will be used during LTRT operation. Included in the instruction is the mode of handling (during LTRT operation) input codes that are out of the range of codes in the table.

This instruction initializes the contents of the global array, which is to be subsequently used in generating a translation table, so that each input code has no processing requirements (no ochr, func, mask, addr, or inlen operands: see the LTRTENT instruction). No constants in the application program are generated by this instruction.

Name	Operation	Operand
[label]	LTRTBEG	[RANGE=([low code],[high code],[mode])]

RANGE
Indicates the range of codes recognized for this translation table, and action to be taken when codes are not within the specified range.

low code: Is the value of the lowest code to be used; low code defaults to zero. The value may be X'nn' (hexadecimal), nnn (decimal), or C'c' (character).

high code: Is the value of the highest code to be used; high code defaults to 255. The value may be X'nn' (hexadecimal), nnn (decimal), or C'c' (character).

mode
Specifies the treatment (during LTRT operation) of codes that are out of the range specified:

ERROR Specifies that an out-of-range code will cause translation to stop with the appropriate condition code. ERROR is the default for mode.

SKIP Specifies that out-of-range codes are to be processed as a NULL input code. Translation is not stopped.

MOVE Specifies that any out-of-range code is to be moved directly to the output area without translation. Translation is not stopped.

use Specifies that the value of *use* will be placed into the output area for any code that is out of range. The *use* value may be X'nn' (hexadecimal), nnn (decimal), or C'c' (character).

LTRTENT--Translate Table Entry

Each operand in the LTRTENT defines the processing requirements for an input code. All input codes may be defined by one LTRTENT instruction or may be described by a number of LTRTENT instructions. The number of operands per instruction for DOS users is limited to 200.

This instruction modifies the contents of the global array that is to be subsequently used in generating a translation table. No constants are generated in application program by this instruction.

Name	Operation	Operand

```
[label] LTRTENT  (code,[ochr],[func],[mask],
                 [addr],[inlen]) [,(...)]
```

code

Is a one-byte character that is to be translated. The value may be X'00' to X'FF' but must also be within the range specified by the LTRTBEG instruction. The specification of code may be X'nn' (hexadecimal), nnn (decimal), or C'c' (character).

If the same code is specified more than once, only the last occurrence will be used in the translation table.

ochr

Is a byte or group of bytes identifying the output for the associated input code. *ochr* may be 1 to 7 bytes of characters (C'cc...c') or 1 to 4 bytes of hexadecimal data (X'nn...n'). If *ochr* is not specified, no ochr will be generated for the input code during LTRT operation.

func

May be one of the following functions to be performed when the associated input code is recognized. The functions value, defined below for each function, for the last input code processed will be returned in the parameter list (TRPFNC) when LTRT completes. If the last input code processed has no associated function, X'00' will be returned.

CASn

Is a shift to the translation table identified by n (1 to 4). The shift will be effective on the next input code. The value in the translation table for CASn is (X'40' +n-1).

DS

Is a downshift to the previously used table. The new table will be effective on the next input code. More than one consecutive DS without an intervening CASn forces a return to translation table 1. The value in the translation table for DS is X'60'.

TW

Specifies that a transparent write (no translation) is to be performed for the number of bytes specified in the first (or second) byte following the input code that caused the TW function. For that length, all input that immediately follows the

length byte will be placed in the output area as received. The input length (see *inlen* below) may be set to V1 (or V2, respectively) for transparent write operations. If *inlen* was not specified, the default is V1. The value in the translation table for TW is X'61'.

BSP

Specifies that the current output text position should be moved back one position nondestructively. The output text position will not be modified if an attempt is made to backspace beyond the beginning of the output area. The value in the translation table for BSP is X'62'.

ADV

Specifies that the current output text position should be advanced one position nondestructively. The value in the translation table for ADV is X'63'.

ufc

Can be specified by the user to be a hexadecimal value (X'01' to X'3F') or a decimal value (1 to 63). This value will be returned to the parameter list when the associated input code is recognized.

mask

Specifies that this input code is a translation break. The mask (nonzero) may be hexadecimal (X'nn'), decimal (nnn), or character (C'c') and must match one or more bits of the current parameter list mask (TRPMSK) to be effective.

addr

Is the label of an AP routine that is to get control when the translation break (see **mask**) is effective. If *addr* is not specified for an effective break code, control goes to the next sequential instruction.

If the label specified does not appear in the current assembly, an EXTRN must exist for the label and the corresponding assembly must contain an ENTRY for the label.

inlen

Specifies length to be associated with the code, *inlen* may be any self-defining term in the range 1 to 255 (X'01' to X'FF'), or it may be V1 or V2.

At operation time, *inlen* of 1 to 255 represents the displacements (in bytes) from the input code currently being processed to the next input code to be processed. The default is 1 (except for TW; see TW above).

Specifying the letter V1 (or V2) for *inlen* indicates that the input code's associated length is variable. The length (0 to 255) is in the first (or second, respectively) byte following the input code, and represents the displacement from the byte following the length byte to the next input code to be processed.

LTRTGEN--Translate Table Configuration

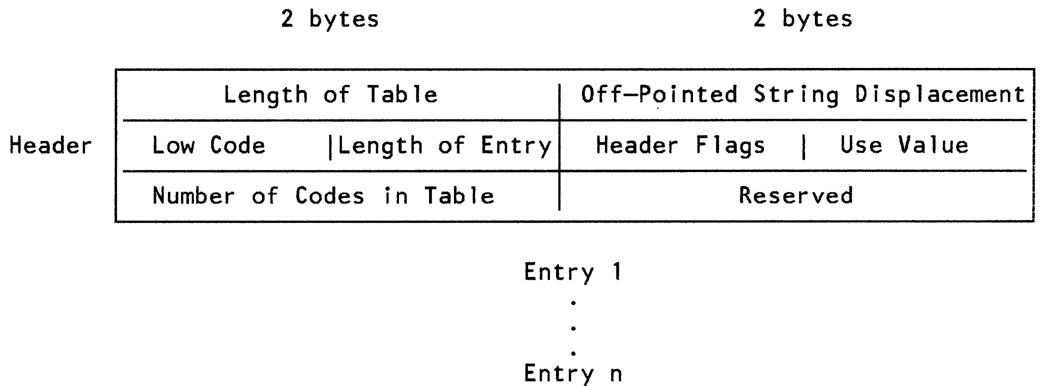
LTRTGEN causes the translation table defined by the preceding LTRTBEG and LTRTENT instructions to become part of the program's data definition.

LTRTGEN uses the assembler global array initialized by LTRTBEG and modified by LTRTENT to create a series of constants that are the translation table.

LTRTGEN has no operands.

Name	Operation	Operand
[label]	LTRTGEN	

The format of the translation table is as follows.



LWAIT--Wait

The LWAIT instruction causes the currently operating station to enter wait state, and therefore become nondispatchable until an ending condition is present.

Name	Operation	Operand

```
[label] LWAIT
```

Condition Codes: None are set

Program Checks: None are set.

Programming Notes: Ending conditions are:

1. an attention, signaled by the operator pressing the keyboard Reset key twice in succession, **or**

2. any asynchronous interrupts for which an entry point has been specified in the application program BEGIN instruction.

When an ending condition occurs, the station becomes dispatchable and operation continues with the next sequential instruction (that is, the instruction following LWAIT) with SMSWAIT set to a one-byte value indicating the ending condition.

The LWAIT instruction could therefore be followed by:

```
SETFPL SMSWAIT
LSEEK 1,label
```

where "label" locates a table with each entry containing an ending condition value and an associated branch address.

The ending condition values are listed below in order of priority (highest priority is first):

Hex value	**Meaning**
10	CPU message pending
20	ALA message pending
30	terminal message pending
40	station message pending
50	program interrupt pending
60	timer interrupt pending
70	attention signaled

If more than one ending condition is present, only the highest priority condition's value is posted.

If any ending conditions are present when the LWAIT instruction is performed, a dispatch cycle is taken before operation continues with the next sequential instruction.

If an interrupt occurs that causes the controller to end an LWAIT and if the station then issues an LEXIT, the controller does not dispatch the station at the asynchronous entry point for that interrupt.

LWRITE--WRITE Station-to-Station Message

LWRITE writes data to another logical work station. Before the LWRITE is issued, the station number must be stored in binary form in SMSDSS.

An LWRITE to another station is a request only. No data is transferred until the receiving station issues an LREAD. The LWRITE is rejected if the receiving station already has a message waiting.

When an LWRITE ST instruction with a length of zero is issued, the receiving station, when it completes the LREAD ST, will receive a condition code of 1, and SMSIML will be set to zero.

Immediate status is returned (X'0401') if the receiving station's application program does not have an AST entry point. The LWRITE still occurs, however, and the status may be ignored if the receiving station's program eventually performs an LREAD.

Name	Operation	Operand

```
                               ⎧ defcon2           ⎫
                               ⎪ defld2            ⎪
[label] LWRITE   ST,           ⎨ (defrf2)          ⎬
                               ⎪ (reg2)            ⎪
                               ⎪ seg2              ⎪
                               ⎩ seg2,disp2,len2   ⎭
```

ST
Specifies a write to another station.

operand 2
Is a field containing the data to be written. If only seg2 is specified, the data is written starting at the location pointed to by the secondary field pointer (SFP) up to, but not including, the location pointed to by the primary field pointer (PFP); otherwise, the data is as addressed. The maximum length of a message to be written is 255 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	The write operation was completed successfully.
02	ST	Status is returned; the status code is contained in SMSDST. (See Appendix E for an explanation of the status codes.)

Program Checks (hex): 01, 02, or 27 can be set.

MASK--Mask (For EDIT Instruction)

MASK produces a data string used by the EDIT instruction to format monetary fields.

Name	Operation	Operand
label	MASK	[fill,] t'characters'

fill

Is a one-byte hexadecimal (X'*xx*'), character (C'*c*'), or binary B'*nnnnnnnn*' value used in place of leading zeros when zero suppression is specified. If this operand is omitted, blanks are used as the fill for zero suppression.

t'characters'

Is the control field in which *t* indicates the type of data: X for hexadecimal or C for character; and '*characters*' is made up of the following characters:

9	Indicates a significant position.
Z	Indicates leading zero suppression.
$	Is a dollar-sign insertion.
b	Indicates blank insertion.

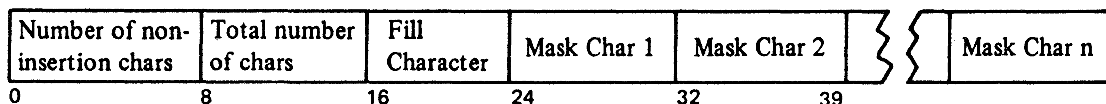
All other characters are insertion characters. These may be any characters except 9, Z, $, and b.

See the EDIT instruction for further description.

Notes:

1. An insertion character cannot be the last character in the mask field.
2. The label of an EQUATE instruction cannot be used as an operand for MASK.

The **MASK** format is:



MOD--Modulus Factor (For MODCHK Instruction)

MOD specifies the type, modulus number, and weighting factors used in the modulus check done by the MODCHK instruction. The number of weighting factors determines the maximum length of the field to be checked.

Name	Operation	Operand
label	MOD	modnum, { wt1 | (wt1,wt2,...,wtn) } [MA | AP | UN]

modnum
Is an absolute number from zero to 99 used as the modulus number in the check.

wt1,wt2,...,wt*n*
Are up to 31 four-character weighting factors in the range of -128 to +127. If only one weighting factor is specified, it must be coded without parentheses.

MA
Specifies a type of modulus check in which each character in the field is multiplied by the corresponding weighting factor, and the sum of these multiplications is divided by the modulus number. This type of modulus check is performed if the type operand is omitted.

((n*1* X wt*1*) + (n*2* X wt*2*) + ...

(n*n* X wt*n*)) / modnum

AP
Specifies a type of modulus check in which each character in the field is multiplied by the corresponding weighting factor, and the numbers comprising each result of the multiplications are divided into two separate values; the numbers in the thousands (K), hundreds (H) and tens (T) positions are used as one value and the number in the units (U) position as another value. The new values from the results are then added together. For example: 5 x 25 = 125 --> 12 + 5 = 17. The sum of all the additions is divided by the modulus number.

n*1* X wt*1* -->K*1*H*1*T*1*U*1*
n*2* X wt*2* -->K*2*H*2*T*2*U*2*
n*n* X wt*n* -->K*n*H*n*T*n*U*n*

((K*1*H*1*T*1* + U*1*) + (K*2*H*2*T*2* + U*2*) + ...(K*n*H*n*T*n* + U*n*)) / modnum

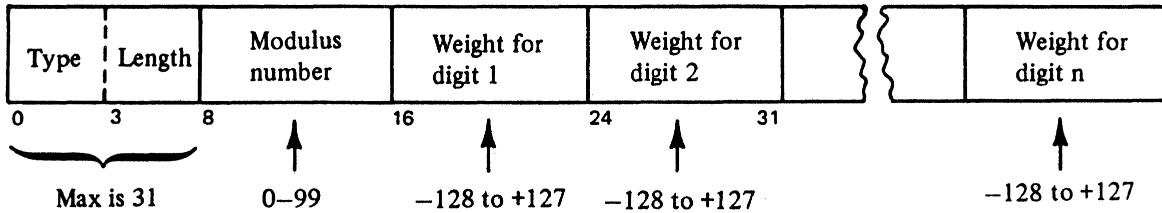
UN

Specifies a type of modulus check in which each character in the field is multiplied by the corresponding weighting factor and the only number kept from each result of the multiplications is the number in the units position. For example, 5 X 25 = 125 --> 5. The sum of these numbers is then divided by the modulus number.

n*1* X wt*1* --> U*1*
n*2* X wt*2* --> U*2*
n*n* X wt*n* --> U*n*
((U*1*) + (U*2*) + ...U*n*)) / modnum

The format of the modulus factor is as follows:

MOD

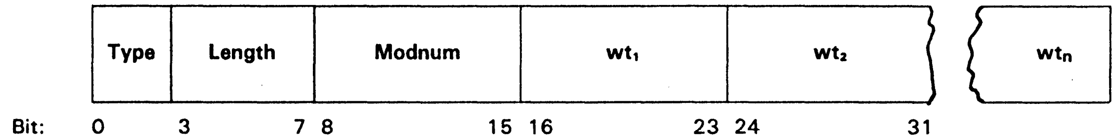


Where:

Type = 000 for UN
= 010 for MA
= 100 for AP

MODCHK--Modulus Check

MODCHK checks a field for valid EBCDIC numeric characters, and that the length is less than or equal to the number of weights. The field is then validated by a modulus check which uses the following parameter list defined by the MOD instruction:



Refer to the MOD instruction for a description of these parameters.

Name	Operation	Operand
[label]	MODCHK	seg1,reg2, { label / (reg3) }

operand 1
Is a field in the specified segment to be checked. The start of the field is indicated by the primary field pointer, and the length by the field length indicator.

operand 2
Is a register (1-15) that will contain the results of MODCHK. If this operand is specified as zero, it does not indicate a register.

operand 3
Points to a MOD instruction or it's equivalent.

label
Is the label of the MOD instruction that defines the parameter list used by MODCHK. The label may be an external symbol when RELOC=Y is specified on the APOPT instruction.

(reg3)
Is the register containing the address of a field with data in the same format that MOD generates.

All modulus checks begin by multiplying each character in the field by the corresponding weighting factor. For example, the first character in the field is multiplied by the first weighting factor listed in MOD. The last step of all the modulus checks is to divide the sum of the results (or variations of the results) of these multiplications by the modulus number (refer to the descriptions of the modulus type operands in the MOD instruction for the different types of modulus checks). If there is no remainder, the field checked is good in terms of the modulus check.

If reg2 specified in MODCHK and the modulus number (*modnum*) are nonzero values, the remainder generated by the modulus check is stored in the reg2. If the modulus number is zero and the register is nonzero, the sum of all the results (or variations of the results) of the multiplications performed in the modulus check is stored in the register. If the register is zero and the modulus number is nonzero, the remainder is not stored. If both the register and modulus number are zeros, no modulus check occurs, and a condition code of hex 01 is set.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	The check was successful, or the register and modulus numbers were zero.
02	IL	The field is too long.
04	NN	The field is not numeric.
08	MD	A modulus error occurred (can be returned only if register is zero).

Program Checks (hex): 01, 02, or 27 can be set.

Programming Note: If you do not specify reg2 as zero, you are indicating a request to do further processing of the result of the modulus check. For example, you may wish to calculate a modulus-check digit.

A condition code of 01 is returned even if there is a remainder. Condition code 08 can be returned only when the specified register is zero.

MPYFLD--Multiply Field

MPYFLD algebraically multiplies the binary contents of a register by the binary contents of a field. The result is placed in the register.

Name	Operation	Operand

```
                                  { defcon2          }
                                  { defld2           }
[label] MPYFLD    reg1,           { (defrf2)         }
                                  { (reg2)           }
                                  { seg2,disp2,len2  }
```

operand 1
Is a register that contains the multiplicand. At the end of the operation this operand contains the product.

operand 2
Is a field that contains the multiplier. The length of the field must be from zero to 6 bytes; if the length is zero, the register is multiplied by zero; if the length is not zero, the leftmost bit in the field is the sign.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overflow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, or 27 can be set.

MPYFLDL--Multiply Field Logical

MPYFLDL multiplies the contents of a register by the contents of a 6-byte field. If the field is less than 6 bytes, it is treated as a 6-byte field by propagating zeros. The binary contents of the register are then multiplied by the binary contents of the field and the result is placed in the register.

Name	Operation	Operand

```
                                  ⎧ defcon2          ⎫
                                  ⎪ defld2           ⎪
[label] MPYFLDL   reg1,           ⎨ (defrf2)         ⎬
                                  ⎪ (reg2)           ⎪
                                  ⎩ seg2,disp2,len2  ⎭
```

operand 1

Is a register that contains the multiplicand. At the end of the operation this operand contains the product.

operand 2

Is a field that contains the multiplier. The length of the field must be from zero to 6 bytes; if the length is zero, the register is multiplied by zero.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overflow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, or 27 can be set.

MPYREG--Multiply Register

MPYREG algebraically multiplies the binary contents of one register by the binary contents of another register. The result is placed in the register indicated by operand 1.

Name	Operation	Operand
[label]	MPYREG	reg1,reg2

operand 1
Is a register that contains the multiplicand. At the end of the operation, this register contains the result.

operand 2
Is a register that contains the multiplier.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overflow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks: None are set.

MPYZ--Multiply Zoned Decimal

This instruction multiplies zoned decimal operand 1 by zoned decimal operand 2, and replaces operand 1 with the result. The length of either operand is 1-63 bytes; an operand over 15 bytes must be selected using register addressing.

Note: This is an optional instruction; module P31 must be specified on the OPTMOD configuration macro.

Name	Operation	Operand

```
                        { defld1            }   { defcon2            }
[label] MPYZ            { (defrf1)          } , { defld2             }
                        { (reg1)            }   { (defrf2)           }
                        { seg1,disp1,len1   }   { (reg2)             }
                                                { seg2,disp2,len2    }
```

operand 1
Is a field containing the zoned decimal multiplicand, and the location of the result. If the result is less than the operand length, each high-order byte is set to X'F0'. The field must not be in Segment 14.

operand 2
Is a field containing the zoned decimal multiplier.

Condition Codes: One of the following can be set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overflow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, 09, or 27 can be set.

MVCZ--Move and Convert Zoned Decimal

This instruction converts a zoned decimal number into displayable form. The zoned decimal number specified by operand 2 is moved to operand 1. The sign (zone bits of the least significant number) is ORed with X'F0' to give the digit a displayable value.

Note: MVCZ is an optional instruction, and requires specifying module P31 on the OPTMOD configuration macro.

Name	Operation	Operand
[label]	MVCZ	{ defld1 | (defrf1) | (reg1) | seg1,disp1,len1 } , { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Is the field into which the zoned decimal data is moved for displaying. If this operand is longer than operand 2, this operand is padded to the left (high-order positions) with X'F0's. If this operand is shorter than operand 2, the high-order digits are truncated. In either case, an appropriate condition code is set. The field must not be in Segment 14.

operand 2
Is a field of decimal numeric zoned data to be moved to operand 1.

Note: Operands 1 and 2 can range 1 to 15 bytes for non-register addressed fields, and up to 63 bytes when register addressing is used.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	EQ	The operands were equal lengths.
04		Operand 1 was padded with X'F0'.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, 03, 09, or 27 can be set.

MVDI--Move Data Immediate

MVDI moves a one- or two-byte immediate operand to a field. At the end of the operation, the primary field pointer of the segment to receive the immediate data points 1 byte past the end of the field (unless register addressing is used), and the field length indicator is not changed.

Name	Operation	Operand
[label]	MVDI	{ defld1 | (defrf1) | (reg1) | seg1,disp1 } , immdata2

operand 1
Is a field into which the immediate data is to be moved. The field must not be in Segment 14.

operand 2
Is one or two bytes of immediate data.

Condition Code: The code is not changed.

Program Checks (hex): 01, 02, 03, or 27 can be set.

MVFLD--Move Field

MVFLD moves the contents of a field to the specified segment, starting at the location specified by the PFP of that segment. The number of bytes to be moved is determined by the specified or implied length of the second operand (the field to be moved) unless the specified length is 0. At the end of the operation, the FLI is unchanged and the PFP, of the segment that received the field, points to the byte following the field.

Name	Operation	Operand
[label]	MVFLD	seg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Is a field in the specified segment to receive the data. The number of bytes can range from 1 to 32 767. If you specify a length of 0 for the second operand, then the FLI of operand 1 determines the number of bytes to be moved. If the field lengths of both fields are zero, no operation takes place. The field must not be in Segment 14.

operand 2
Is a field that is to be moved. This field can be from 1 to 15 bytes long unless you specify register addressing, which allows a length ranging from 1 to 32 767 bytes.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, or 27 can be set.

| MVFLDR--Move Field Reverse

MVFLDR moves the contents of a field to a segment, starting at the location specified by the PFP of that segment. The number of bytes to be moved is determined by the specified or implied length of the second operand; that is, the field to be moved. At the end of the operation, the FLI is unchanged and the PFP, of the segment that received the field, points to the byte following the field.

MVFLDR reverses the order of the source data so that the first byte in the source field becomes the last byte in the target field.

MVFLDR requires the P60 optional module specified by the OPTMOD macro in your system configuration.

Name	Operation	Operand

```
                                { defcon2          }
                                { defld2           }
[label] MVFLDR     seg1,        { (defrf2)         }
                                { (reg2)           }
                                { seg2,disp2,len2  }
```

operand 1
: Is a field in the specified segment to receive the data. The number of bytes can range from 1 to 32 767. If you specify a length of 0 for the second operand, then the FLI of operand 1 determines the number of bytes to be moved. If the field lengths of both fields are zero, no operation takes place. The field must not be in Segment 14.

operand 2
: Is a field that is to be moved. This field can be from 1 to 15 bytes long unless you specify register addressing, which allows a length ranging from 1 to 32 767 bytes.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, 09, or 27 can be set.

MVFXD--Move Fixed

MVFXD moves the contents of a field to another field. The number of bytes to be moved is determined by the specified or implied length of the second operand; that is, the field to be moved. At the end of the operation, the FLI is unchanged and the PFP, of the segment that received the field, points to the byte following the field.

Name	Operation	Operand
[label]	MVFXD	{ lefld1 | (defrf1) | (reg1) | seg1,disp1 } , { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Is a field that receives the operand 2 data. The length of this field is ignored. The field must not be in Segment 14.

operand 2
Is the field to be moved. The length of this field determines how many bytes are moved. The length can be from 1 to 255 unless you specify register addressing, which allows a length ranging 1 to 32 767. If the field length is zero, no operation results.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, or 27 can be set.

Programming Notes: For example, a savings deposit is either transmitted to the host processor to update the data base or journaled on the diskette for later transmission. The fields required in either case are:

transaction code/accountno./amount

The following example shows the instructions that build the transaction code and account number portions of the output message.

```
OUTFLD DEFLD    OUTSEG,0,19                  1
SAVDEPCD DEFCON C'01'                        2

SAVRTN                                       3
                MVFXD OUTFLD,SAVDEPCD        4
                MVSEG OUTSEG,INPUTSEG        5
                MVSEG OUTSEG,INPUTSEG        6
```

1. Defines the entire output field.
2. Defines the transaction type code for a savings deposit as 01. This avoids storing or transmitting the 6-byte code entered by the teller.
3. (Miscellaneous processing)

4 Moves the transaction type code to the first two bytes of OUTFLD. The PFP changes, when the MVFXD is operated, to point to the third byte of OUTFLD.

5 Moves the account number to OUTFLD.

6 Moves the amount.

| MVFXDR--Move Fixed Reverse

MVFXDR moves the contents of a field to another field. The number of bytes to be moved is determined by the specified or implied length of the second operand; that is, the field to be moved. At the end of the operation, the FLI is unchanged and the PFP, of the segment that received the field, points to the byte following the field.

MVFXDR reverses the order of the source data so that the first byte in the source field becomes the last byte in the target field.

MVFXDR requires the P60 optional module specified by the OPTMOD macro in your system configuration.

Name	Operation	Operand
[label]	MVFXDR	{defld1 | (defrf1) | (reg1) | seg1,disp1} , {defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2}

operand 1
Is a field that receives the operand 2 data. The length of this field is ignored. The field must not be in Segment 14.

operand 2
Is the field to be moved. The length of this field determines how many bytes are moved. The length can be from 1 to 255 unless you specify register addressing, which allows a length ranging 1 to 32 767. If the field length is zero, no operation results.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, 09, or 27 can be set.

MVSEG--Move Segment

MVSEG moves the contents of a segment-header addressed field to another segment-header addressed field. The number of bytes to be moved is determined by the FLI of the second operand; that is, the field to be moved.

If the move is between two locations within the same segment, the SFP is used as the beginning of the receiving field. At the end of the operation, the FLI is uncnanged and the PFP points to the byte following the field.

If the move is between different segments, the PFP of the receiving segment is used as the beginning of the receiving field. At the end of the operation, the PFP points one byte past the end of the receiving field; the PFP of the segment containing the original field is not changed.

In both cases, the SFP and FLI of the segment containing the data to be moved are unchanged.

Name	Operation	Operand
[label]	MVSEG	seg1,seg2

operand 1
Is a field in the specified segment to receive the data. The field must not be in Segment 14.

operand 2
Is a field in the specified segment whose contents are to be moved. The location of the field to be moved is determined by the PFP, and the number of bytes to be moved is determined by the FLI. The field length can be 1 to 32 767 bytes. If the field length is zero then no operation results.

Condition Codes: The code is not changed.

Program Checks (hex): 01 or 02 may be set.

MVSEGR--Move Segment Reverse

MVSEGR moves the contents of a segment-header addressed field to another segment-header addressed field. The number of bytes to be moved is determined by the FLI of the second operand; that is, the field to be moved.

If the move is between two locations within the same segment, the SFP is used as the beginning of the receiving field. At the end of the operation, the FLI is unchanged and the PFP points to the byte following the field.

If the move is between different segments, the PFP of the receiving segment is used as the beginning of the receiving field. At the end of the operation, the PFP points one byte past the end of the receiving field; the PFP of the segment containing the original field is not changed.

In both cases, the SFP and FLI of the segment containing the data to be moved are unchanged.

MVSEGR reverses the order of the source data so that the first byte in the source field becomes the last byte in the target field.

MVSEGR requires the P60 optional module specified by the OPTMOD macro in your system configuration.

Name	Operation	Operand
[label]	MVSEGR	seg1,seg2

operand 1
Is a field in the specified segment to receive the data. The field must not be in Segment 14.

operand 2
Is a field in the specified segment whose contents are to be moved. The location of the field to be moved is determined by the PFP, and the number of bytes to be moved is determined by the FLI. The field length can be 1 to 32 767 bytes. If the field length is zero then no operation results.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, 09, or 27 can be set.

OVLYSEC--Define Load Address and Entry Point

OVLYSEC defines the load address and entry point of overlay sections assembled under the nonrelocatable and relocatable assembler options (RELOC=N or RELOC=Y specified in the APOPT instruction). OVLYSEC must be the first instruction in the overlay section.

In relocatable and nonrelocatable assemblies, the OVLYSEC instruction causes an ADDMEM command to be generated for use by the Host Support. For relocatable and nonrelocatable assemblies, the label of the OVLYSEC instruction is used to name the CSECT produced.

For nonrelocatable assembly, the load address can be expressed symbolically only when the root and overlay sections are assembled together. Otherwise, the load address must be stated as an absolute location in Segment 14.

For relocatable assemblies, the load address can be expressed symbolically regardless of whether the root and overlay are assembled together or separately.

Name	Operation	Operand

```
                          { org                             }
label    OVLYSEC  ORIGIN= { ([inst-org] [,const-org])       }

                  [,OVLYEP=entry]

                  [ ,VERSION= { version } ]
                  [           { 0       } ]

                  [ ,INSTR= { N } ]  [,INSNAME=name]
                  [         { Y } ]
```

(Defaults are underlined in the original: 0 for VERSION, Y for INSTR.)

org
inst-org
const-org

The **org** parameter is used to specify the load address for an overlay being assembled with the nonrelocate or relocate options in effect.

The **inst-org and const-org** operands are used only for split application programs. See Appendix F for further information.

These parameters may be specified as an asterisk (*), the **label** of a STOVLY instruction, or a decimal **number**, as follows:

* indicates that the section, for relocatable assemblies, is to be loaded at the end of the root or other overlay section already in storage. For nonrelocatable assemblies, an * indicates that the current value of the assembler instruction counter is to be used as the load address.

label is the label of a STOVLY instruction that defines an overlay load point. For relocatable assemblies, the STOVLY instruction and the OVLYSEC instruction referring to it can be in the same or separate assemblies; for nonrelocatable assemblies STOVLY and OVLYSEC must appear in the same assembly.

number is the absolute location, expressed in decimal, in Segment 14 at which the overlay section is to be loaded. This specification *must be used* when the nonrelocatable assembler option is specified and the root and overlay are being assembled separately.

entry

Is the label of the instruction that is to be used as an entry point. If this operand is omitted, the entry point is the first byte after the 2-byte header in the overlay section.

Note: Do not specify an entry point if you load the overlay using the "expanded" LLOAD instruction. In this case, use the default entry point at the first byte following the two-byte header.

version

Is the version number (0 to 255) of the overlay section.

INSTR

This operand is used only for split application programs. See Appendix F for further information.

INSNAME

This operand is used only for split application programs. See Appendix F for further information.

Condition Codes: The code is not changed.

Program Checks (hex): None can be set.

PAKFLD--Pack Field

PAKFLD changes a hexadecimal EBCDIC number (for example: C'012F' which equals X'F0F1F2C6') in a field to a packed value (X'012F' from the preceding example) and places the packed value in a segment-header addressed field. In the packed expression, each 4 bits represents a digit of the EBCDIC number. If an odd number of digits are packed, the leftmost 4 bits of the leftmost byte in the packed field are set to zeros.

Notes:

1. This packed expression cannot be used in arithmetic operations.

2. The length of the packed field will be one-half the length of the unpacked field.

The hexadecimal representations and their packed equivalents are:

Digit	Hexadecimal Representation	Packed Equivalent (4 bits)
0	F0	0 (0000)
1	F1	1 (0001)
2	F2	2 (0010)
3	F3	3 (0011)
4	F4	4 (0100)
5	F5	5 (0101)
6	F6	6 (0110)
7	F7	7 (0111)
8	F8	8 (1000)
9	F9	9 (1001)
A	C1	A (1010)
B	C2	B (1011)
C	C3	C (1100)
D	C4	D (1101)
E	C5	E (1110)
F	C6	F (1111)

Name	Operation	Operand
[label]	PAKFLD	seg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

operand 1
Is a field in the specified segment to receive the packed data. The segment number cannot be 14. The location of the field to contain this packed value within operand 1 is indicated by the primary field pointer (PFP). The field length indicator is ignored unless the length of the unpacked field is zero. Following the operation of the instruction, the PFP points to the first byte past the end of the packed field.

operand 2

Is a field to be packed. If you use register addressing, the operand length can range from 0 to 255 bytes; if you use fixed field addressing, the length ranges 0 to 15 bytes. If the length is 0, the length indicated by the field length indicator of *seg1* is used.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, 03, or 27 can be set.

PAKSEG--Pack Segment

PAKSEG changes a hexadecimal number (for example, C'012F' which equals X'F0F1F2C6') in a segment-addressed EBCDIC field to a packed value (X'012F') and places it in a segment-header addressed field. In the packed expression, each 4 bits represents a digit of the hexadecimal number. If an odd number of digits are packed, the leftmost 4 bits of the leftmost byte in the packed field are set to zeros.

Notes:

1. This packed expression cannot be used in arithmetic operations.
2. The length of the field to contain the packed value will be one-half the length of the field that contains the value to be packed.

Refer to the table under PAKFLD for the hexadecimal values and their packed equivalents.

Name	Operation	Operand
[label]	PAKSEG	seg1,seg2

operand 1
Is a field in the specified segment to receive the packed field. The segment number cannot be 14. If the operands refer to different segments, the location within this segment is indicated by the primary field pointer (PFP). If both operands refer to the same segment, the location of the packed field is indicated by the secondary field pointer; when the operation is completed, the PFP points to the first byte past the end of the packed field and the SFP is unchanged. In either case, the field length indicator is ignored.

operand 2
Is a field in the specified segment to be packed. The PFP indicates the start of the field, and the field length indicator gives the length (0 to 255) of the field containing the value to be packed. If the length is zero, no operation occurs. When the operation is completed, the PFP is unchanged.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, or 03 may be set.

Programming Notes: For example, if the processed teller transaction is packed before being written to the diskette, it requires only half the space normally needed. The following shows the instructions used to pack the output field. The field before being processed is:

X'F0F1F3F3F8F2F2F0F6F5F2C1C4C4'

The packed field is:

X'01338220652ADD'

In this example, the source and result fields overlap so that result data does not replace source data yet to be packed.

```
OUTFLD DEFLD   OUTSEG,0,14        1
       •
PAKRTN SETFPL OUTFLD              2
       SETSFP OUTFLD              3
       PAKSEG OUTSEG,OUTSEG       4
```

1 Defines the source field.
2 Sets the PFP and FLI of OUTSEG to describe the source field.
3 Sets the SFP to the beginning of OUTFLD.
4 Packs the data in place. After operation, the leftmost 7 bytes of OUTFLD contain the packed data.

The pack function can also be used to allow the teller to enter hexadecimal information; for example, the control operator could be told to enter the parameter list used to start the communication link.

PAUSE--Suspend Processing

PAUSE suspends processing for the logical work station currently in control. Processing resumes with the next sequential instruction when all other stations have had an opportunity to process.

Name	Operation	Operand
[label]	PAUSE	

Condition Codes: The code is not changed.

Program Checks: None are set.

PLPCMD--Post-List Processor Commands

The PLPCMD instruction generates commands for execution by the post-list processor when editing the listing produced by the assembler.

Name	Operation	Operand
	PLPCMD	{ EDIT NOEDIT DELETE NODELETE }

EDIT
Resumes post-list processing of assembled output.

NOEDIT
Temporarily stops post-list processing of assembled output. When the post-list processor encounters this in the assembled program, the assembled output is passed unprocessed to the output listing. A PLPCMD instruction with the EDIT operand resumed post-list processing of the assembled output lines.

DELETE
Causes the post-list processor to delete all further assembler output lines from the output listing.

NODELETE
Causes the post-list processor to resume normal post-list processing of output lines. This restores the processing deleted by the DELETE operand.

PRINTI--Print Macro Expansion

The PRINTI instruction enables you to print the assembler instructions (labels, mnemonics, and operands only), in the Post List Processor (PLP) listing, that result from expansion of user-written macros found in the application program. To print these instructions, PRINTI ON must be coded before the desired macro is encountered or be the first statement in the macro definition. The ON state remains in effect until PRINTI OFF is coded. When PRINTI ON is coded, all assembler instructions will be printed regardless of the degree of macro nesting. All assembler instructions encountered in open code will be printed twice when PRINTI ON is in effect.

Name	Operation	Operand
[label]	PRINTI	[ON|OFF]

For Example:

4700 assembler instructions
•

PRINTI ON	Macro
USERMAC	Assembler instructions (printed once in
PRINTI OFF	PLP listing)
•	Mend

•
4700 assembler instructions (not printed in PLP listing)
•
•

	Macro
USERMAC	PRINTI ON
•	Assembler instructions (printed once in
•	PLP listing)
•	PRINTI OFF
•	Mend

•
PRINTI ON
•
4700 assembler instructions (printed twice, encountered in open code)

Note: The PRINTI macro instruction prints only the label field, operation-code field, and operand field. (No comment fields are printed.) Only the first 112 characters of the operand field are printed; remaining characters are omitted. PRINTI does not support the BEGIN macro. Omitted operands are not printed. For example:

DEFLD 2,,0 -- if coded, will print:
DEFLD 2,0

The following macro instructions issue a PRINTI ON instruction upon entry and issue a PRINTI OFF instruction on exit:

APBDUMP
DEFDMP

REBASE--Restore the Base Register for a DSECT

The REBASE instruction restores the base register for the specified DSECT to the register value previously saved by a SAVEBASE instruction.

Name	Operation	Operand
	REBASE	label

label
The label of an LDSECT instruction.

SAVEBASE--Save the Base Register for a DSECT

The SAVEBASE instruction saves the current base register value of the specified DSECT in a global symbol table. This value was specified either at definition time with the BASE keyword on the LDSECT instruction or by the later specification of a base register with the USEBASE instruction.

Name	Operation	Operand
	SAVEBASE	label

label
The label of an LDSECT instruction.

SCALE--Scale Number

The SCALE instruction verifies and formats a relatively free-form character-type number.

Inputs for the SCALE instruction are:

1. A relatively free-form character-type input area which may have a one-byte header field preceding the number field. The input area is unchanged by the execution of the SCALE instruction.

2. A parameter list defined by a COPY DEFSCA instruction.

Outputs for the SCALE instruction are:

1. The formatted character-type number in an output area.

2. The PFP of the segment containing the output area points to the first byte of the output area. The FLI is set to the length of the output area. If the fixed field is indexed, then the new setting of the primary field pointer will include the index value.

The parameter list contains the following fields (see COPY DEFSCA in Appendix B) and must contain the following information:

SCALEN -- 1-byte field defining the length of the output area in bytes. The output area immediately follows the parameter list.

SCACHR -- A 1-byte field containing, in EBCDIC, the input data character to be treated as a decimal place character (scale character). If the input data does not contain a scale character, one is assumed immediately following the rightmost character of the input area. If a scale character appears more than once in the input area, only the rightmost occurrence is treated as *the scale character.* The scale character is not placed into the output area.

SCAFAC -- A 1-byte scale factor defining the number of times the input number will be multiplied by 10 before being placed into the output area. Each multiplication by 10 effectively shifts the scale character 1 character position to the right. The input area is not changed by this operation.

SCAINP -- A 1-byte input flag indicating which of the six special header characters defined in the SCAHDR field will be placed into the leftmost byte of the output area.

Bits 0-5 -- Mask on (1) or off (0) special header characters 1-6 (respectively) in the SCAHDR field, as described in SCAHDR below.

Bits 6 and 7 -- Reserved and must be set to 0.

SCAHDR -- A 6-byte field containing six special header characters (SHCs). If the first character of the input area equals any of these special header characters, the input area is said to contain an optional header field. The header field character will be placed in the leftmost byte of the output area if the corresponding flag bit in the SCAINP field is on. If the corresponding bit is off, the header field is ignored.

SCADEL -- A 4-byte field containing four delete characters. Delete characters are those characters that are considered valid but are to be deleted from positions to the left of the scale character before the number appears in the output area. If a scale character appears more than once in the number field of the input area, the rightmost scale character (*the scale character*) will be deleted as described under SCACHR above. Other scale characters will be deleted if the scale character is also a delete character, otherwise they will be placed in the output area.

SCARES -- A 3-byte reserved field that must be set to 0.

SCASIG -- A 1-byte field used by the SCALE instruction to inform the application program of the number of significant digits in the output area. This number is determined by:

- The total bytes from the leftmost nonzero character (excluding the header character, if present) to the rightmost position of the output area; or
- The scale factor, whichever is larger.

Note: To use this instruction, you must code the P68 operand on the OPTMOD configuration macro.

Name	Operation	Operand
[label]	SCALE	seg1, { defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 1

Is a field in the specified segment defining the input area. The PFP and FLI define the location within the segment and the field length.

operand 2

Is a field containing the parameter list. The length implied is ignored as the parameter list has a fixed length. The field must not be in Segment 14. The output area immediately follows the parameter list. The length of the output area is specified in the parameter list (SCALEN).

Note: The input and output areas should not overlap. If they do overlap, the results are unpredictable.

Condition Code: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	SCALE was successful.
02		An underflow occurs.
04		An overflow occurs.
08		Invalid input data.

Program Checks (hex): 01, 02, 09, 11, or 27 can be set.

Note: Program check 11 (invalid parameter list) can be set by:

a. SCALE factor greater than output area length.

b. Reserved bits (6,7) of SCAINP are not zero.

Programming Notes: The following shows how various input forms (in dollars and cents) could be reformatted to scale character-type numbers:

Input at Terminal	SCALE Output Area
$12.3	0001230
13,000	1300000
1.26	0000126
-.6	-000060
+2	0000200

If arithmetic operations are to be performed, the application could convert the scaled character-type number to a binary number using the LDSEGC instruction.

An example of how this instruction is used to produce scaled numbers in the instruction's output area is shown below.

```
NOTOK     EQUATE    X'0E'

DEFSCAS   EQUATE    3                                          1

   COPY   DEFSCA                                               2

FORMAT    DEFCON    X'07',C'.',X'02',B'01000000',              3
                C'+-$$$$',C',,,,',X'000000',X'00'
          •
          •
          •
  MVFXD   SCAPAR,FORMAT                                        4
          •
          •
          •
```

Read the input number from the keyboard into Segment 2, set the PFP to the first byte of the number, and set the FLI to the length of the number. For example:

seg2 $12.3

PFP FLI

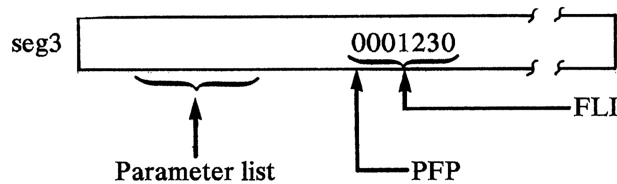
```
SCALE   2,SCAPAR                                  5
JUMP    NOTOK,ERROR                               6
LDSEGC  4,3                                       7
```

1 Defines Segment 3 as the location of the SCALE parameter list (required for the COPY DEFSCA instruction).
2 Copies the SCALE parameter list (SCAPAR) into the code.
3 Defines the values for the SCALE parameter list as follows:

Value	Parameter List Field	Meaning
X'07'	SCALEN	Output area is 7 bytes.
C'.'	SCACHR	Scale character is a decimal point.
X'02'	SCAFAC	Move decimal point two places to right and fill with character zeros if necessary.
B'01000000' C'+-$$$$'	SCAINP } SCAHDR }	Minus (–) sign is to be recognized as a special header character and is to be placed in the leftmost position of the output area when it appears in the leftmost position of the input area.
C',,,,'	SCADEL	Delete commas to the left of scale character.
X'000000'	SCARES	Must be set to zeros.
X'00'	SCASIG	Initializes SCASIG to zero for SCALE to update with number of significant digits.

4 Initializes SCALE parameter list.

5 Issue SCALE instruction to place formatted character-type number in the output area immediately following the parameter list. Following the execution of the instruction, the PFP and FLI are set as follows:



6 Tests the condition code following the execution of the SCALE instruction. If CC=02, 04, and/or 08, branch to an ERROR subroutine.

7 Loads register 4 with the binary equivalent of the character type number placed in the output area of the parameter list, and addressed by the PFP and FLI in Segment 3.

When using the SCALE instruction, the condition code is set following the execution of the instruction to identify various error conditions. The error conditions are:

- underflow
- overflow
- invalid input data

An underflow condition results from truncating an input value to the right of the scaled character. This condition can be the result of either incorrectly entered data or the improper specification of the scale factor (SCAFAC). An example of incorrectly entered data is:

Input	SCALE Output
1.234	0000123

The input is incorrect because the program does not expect a three-place decimal value (SCAFAC = X'02') and places the scaled character-type number right-justified in the output area after moving the decimal place two character positions to the right. If, in the example, SCAFAC was specified as X'01', all input values having two or more characters to the right of the scale character would be truncated.

Overflow can result if the output area, specified by SCALEN, is too small. The output area must be sufficient to accept the largest input value plus a special header character, if specified. The following is an example of overflow; note that the most significant digit is dropped, not the special header character:

Input	SCALE Output
-13,526.35	-352635

The SCALE instruction also validity-checks the input for non-numeric characters. If an input character is non-numeric and is not a special header character or delete character, the data is moved to the output area as though it were valid, but the condition code is set to indicate invalid data. For example:

Input	**SCALE Output**
ABC.DE	00ABCDE

The condition code setting is cumulative. For example, a code of 06 indicates that both an overflow and underflow condition occurred.

SCRPAD--Scratch Pad

This instruction initializes the Scratch Pad Area (SPA) allocated during system configuration and adds, replaces, or deletes SPA elements.

Before issuing SCRPAD, you must create a control parameter list using COPY DEFSCP. You must then set the parameter list fields to define the SCRPAD operation and identify the existing element and/or the new element value, as described under Operand 2.

Optional module P2A must be included in the system configuration OPTMOD macro to use this instruction.

The SCRPAD instruction has the following format:

Name	Operation	Operand
[label]	SCRPAD	{ defld2 | (defrf2) | (reg2) | seg2,disp2 }

Operand 2

Is an area containing the parameter list defined by DEFSCP. Segment 14 cannot be used. Any length specified by this operand is ignored; the COPY DEFSCP instruction determines the parameter list length. The fields in the parameter list are used to provide information for the various SCRPAD functions. When you use register addressing to locate a parameter list, the parameter list can be located in a noncurrent segment space. However, if the parameter list contains the address (segment, displacement) of other storage areas (that is, input/output areas, tables) the other storage areas are always in the current segment space. The fields set by the application program or returned by SCRPAD are:

SCPFC: A one-byte field defining the SCRPAD operation. This field must always be completed.

Hex Code	Explanation
01	Initialize the SPA.
02	Add an element (elements are added sequentially).
03	Replace an element using either key or number.
04	Retrieve an element using either key or number.
05	Retrieve an element for updating.
06	Delete an element using either key or number.
07	Delete all elements.
08	Exchange an element with a data area.
23	Add or replace a keyed element.
25	Retrieve or add a keyed element.

SCPTYP: This one-byte field determines whether SCRPAD searches SPA elements by key, or by number.

Hex Code	Explanation
00	Access by key.
01	Access by element number.

SCPSPID: A one-byte field containing the ID of the SPA. This value is the hexadecimal equivalent of any value in the range from 0 to 255.

SCPELMN: A 2-byte field containing the number of the element. When accessing by element number, the application program stores the element number in this field. When accessing by key, the SCRPAD instruction places the element number of the keyed element in this field.

SCPKEYL: A 1-byte field containing the length of the key field in the element. The key is always part of the element's data and begins in the first position of the element. Using this field, different application programs could use different length keys to locate the same element. For example, one program could use the first four bytes as a key while another program could use the first six bytes.

SCPDATPT: A 3-byte field containing the location of a data area for use by SCRPAD. The first byte contains the number of the segment; the second and third bytes contain the displacement into the segment. For an add or update request, SCRPAD copies the contents of this data area into the SPA element. For a retrieval request, the retrieved element's data (with key) is copied to this data area. The length of this data area is implied by the element length for the SPA.

SCPELMLN: A 2-byte field containing the length of an element for initializing a SPA. All elements in the SPA are initialized to this element length. This field is used only for SPA initialization.

SCPELMNB: A 2-byte field containing the number of initialized elements in the SPA. This field is completed by the SCRPAD instruction after an initialization request is processed, and indicates the number of elements in the SPA.

Condition Codes: SCRPAD sets both the condition code and a return code in the SCPRC field of the parameter list. The possible condition codes are:

Hex Code	Possible Mnemonic	Explanation
01	OK	SCRPAD executed successfully.
02	ST	Operation was not completed or completed conditionally.

SCPRC: A one-byte field containing a return code indicating the result of the SCRPAD operation.

If the condition code is X'01', the possible return codes and their meanings are:

Hex Code	Explanation
00	SCRPAD executed successfully. If SCPFC was X'23' or X'25', a new element was added.
01	Execution was successful, the element was returned at the SCPDATPT location.

If the condition code is X'02', the possible return codes and their meanings are:

Hex Code	Explanation
02	The SPA is full.
08	Update is pending by another application program to the element selected by SCRPAD. This element is unavailable until that operation ends.
10	Your program tried to add a duplicate element.
20	The specified element was not found in the SPA.
40	The SPA you selected has not been initialized.
80	Invalid SPA ID.

Program Checks (hex): 01, 02, 09, or 11 can be set.

Programming Notes:

Initializing the SPA: After a SPA has been defined during configuration, an application program issues the SCRPAD instruction (using the DEFSCP parameter list) to initialize the SPA to elements of a selected fixed length. The DEFSCP parameter list is completed by the program to contain:

SCPFC X'01' to initialize a SPA.

SCPSPID The ID of the SPA.

SCPELMLN The element length to be used for all elements in the SPA. The length specified should not include the one-byte flag field associated with each SPA element.

Upon completion, field SCPELMNB contains the number of elements in the initialized SPA. Field SCPRC contains the return code.

For example, the following code could be used to initialize SPA ID 01 to contain 9-byte entries:

```
DEFSCPS EQUATE 3                  SCRATCH PAD PARAMETER SEGMENT
        COPY   DEFSCP             COPY PARAMETER LIST FIELDS
        •
        •
        •
        MVDI   SCPFC,SCPINTR      SET TO INDICATE INITIALIZE
        MVDI   SCPSPID,X'01'      SET SPA ID
        MVDI   SCPELMLN,AL1(9)    ALLOCATE 9 BYTE ELEMENTS
        SCRPAD SCPSTR             EXECUTE SCRATCH PAD INSTRUCTION
        BRAN   OK,CONT            CHECK IF OK
        •
*ERROR TESTING
```

```
Element
Number
 1   unassigned

 2   unassigned

    •
    •

 25   unassigned

     unused 6 bytes
```

Adding SPA Entries: After a SPA is initialized, application programs can begin to add new elements. Elements can be added only if element spaces are available; elements are always added at the first unassigned element space. If no unassigned element spaces remain, the element is added at the first space where an element has been deleted.

You can add elements with or without keys. If keys are used, the key must begin in the first byte of the element. When the new element is added, all existing elements are searched to ensure that the new element does not contain a key already used in the SPA. The new element is added only if the key is not a duplicate. If the element is added without key addressing, the search is not made. Note that if an element is added by position, *but* the first characters duplicate the key of another element, problems can occur when the SPA elements are later searched by key.

To add an element, complete the DEFSCP parameter list with the following information:

SCPFC X'02' indicates the add function.

SCPSPID The ID of the SPA.

SCPTYP Set to X'00' to add a keyed element.
Set to X'01' to add a nonkeyed element to be retrieved by element number.

SCPDATPT Set to the location (segment followed by displacement) of the data to be placed in the new element.

SCPKEYL Set to indicate the length of the key, if the element contains a key. This field is used only if field SCPTYP is set to X'00' for keyed addressing.

After SCRPAD completes, field SCPELMN contains the element number of the added element; the return code in field SCPRC is set to indicate the status of the operation.

For example, to add element 1 to SPA01 (element 1 will contain the 4-byte key '1234' and the data 'ELM01'), the coding might be:

```
                •
DATADR     DEFCON                   AL1 (5), AL2 (0) LOCATION OF THE
                                    DATA AREA
                •
BLDDATA    DEFCON                   C'1234ELM01' ELEMENT DATA
                •
           MVDI  SCPFC,SCPADDR      SET FOR AN ADD FUNCTION
           MVDI  SCPSPID,X'01'      SPA ID
           MVDI  SCPKEYL,AL1(4)     SET KEY LENGTH
           MVDI  SCPTYP,SCPKEYM     SET FOR KEYED ADDRESSING
           MVDI  SCPDATPT,DATADR    POINT TO THE ELEMENT DATA
           SCRPAD SCPSTR            EXECUTE THE SCRATCH PAD
           BRAN  OK,CONT
                •
      ERROR ANALYSIS
                •
CONT       EQUATE                   * CONTINUE PROCESSING
```

Element Number	
1	1234ELM01
2	unassigned
	•
	•
25	unassigned
	unused 6 bytes

Retrieving a SPA Element: Use the SCRPAD instruction with the DEFSCP parameter list to retrieve an element from the SPA using either the key or the element number. Some elements in the SPA may contain keys, while others do not. If a search is made for retrieving a keyed element, all elements are assumed to contain keys. SCRPAD searches the beginning of each element, beginning with the first, until a match is found. As shown under "Adding SPA Entries" an unkeyed element may contain characters that during retrieval may be mistaken for a key. SCRPAD retrieves the *first* element whose beginning characters match the key being sought.

To retrieve a SPA element, complete these fields in the parameter list:

SCPFC X'04' retrieves a SPA.

SCPSPID The ID of the SPA.

SCPTYP Hex 00 to search by key, hex 01 to retrieve a specific element by element number.

SCPDATPT Contains the segment number and displacement of a data area where SCRPAD will store the retrieved element. If the search is by key, the desired key must be placed in the beginning bytes of the data area before SCRPAD is issued. On completion, SCRPAD replaces the search key with the retrieved element and its data.

SCPKEYL Set to the number of bytes to be used for the key when an element is to be retrieved by key. If this field is set to '4' for example, SCRPAD will compare the first 4 bytes of the data area to the first 4 bytes of each SPA element. When a match is found, the element is retrieved.

SCPELMN Set to the element number when an element is to be retrieved by element number (and not by key).

When SCRPAD completes, the data area whose location is in SCPDATPT will contain the retrieved element. If keyed addressing is used, the element number is stored in field SCPELMN. The return code in field SCPRC is set to indicate the results of the operation.

Exchanging SPA Elements: Use the SCRPAD instruction and the DEFSCP parameter list to exchange the contents of a SPA element with the contents of a specified data area. To use this function, code the DEFSCP fields as follows:

SCPFC X'08' indicates the exchange function.

SCPSPID Indicates the ID of the SPA.

SCPTYP X'00' indicates addressing by key;
X'01' indicates addressing by element number.

SCPDATPT Contains the segment number and displacement of the data area containing the data to be exchanged. If keyed addressing is indicated, the first bytes of the data should contain the key.

SCPKEYL Indicates the number of bytes in the data to be used as the key, beginning with the first data byte.

SCPELMN Indicates the element number when addressing is by element number.

On completion, the contents of the data area and the contents of the SPA element are exchanged. If addressing is by key, the element number of the exchanged element is returned in SCPELMN.

Retrieving a SPA Element for Update: Use the SCRPAD instruction and the DEFSCP parameter list to retrieve an element, set it in "update-pending" state, modify the element, and replace the element in the SPA. The SCRPAD instruction is first used, just as described in the previous "Retrieving a SPA Element" section, except that the SCPFC field is set to X'05'. Otherwise, the parameter list settings are identical. When SCRPAD is complete, the data area contains the retrieved element. If keyed addressing was used for retrieval, the element's number is stored in SCPELMN. The return code is set in field SCPRC.

The retrieved element is then set in "update-pending" state so that no other request can alter the element until the update is complete.

Replacing a SPA Element: The SCRPAD instruction and the DEFSCP parameter list are used to replace an element either by keyed address or by element number.

If the element to be replaced is in the update-pending state (Retrieve for Update has been issued), only the station that retrieved the element can update it. Otherwise, any station can update the element.

When the element is retrieved and is in the data area, the application can make any desired change. Note that if the beginning characters are changed to duplicate an existing element key, errors can occur when later searching the SPA by key sequence.

To use this function, the DEFSCP fields are set as follows:

SCPFC X'03' indicates replacement.

SCPSPID Set to the ID of the SPA.

SCPTYP X'00' indicates that keyed addressing will later be used to retrieve this element. The beginning data characters form the key; X'01' indicates that this element will later be retrieved by element number.

SCPDATPT Contains the segment and displacement of the data area containing the element data to store into the SPA. This is the replacement data.

SCPKEYL Contains the number of beginning characters that constitute the key. This field is used only for addressing by key (field SCPTYP is set to X'00').

SCPELMN Contains the element number of the element being replaced. This field is used only for replacement using the element address (SCPTYP set to X'01').

If the element is being replaced by key, the first characters (defined by the key length) are used to search through the SPA for a matching key. When a match is found, the data in the data area replaces the data in the element.

For example, suppose the program is to retrieve the element with key '3678', and replace the element data portion with the characters 'CHG01'. The following might be coded:

```
DESKFY   DEFCON C'3678'              DESIRED KEY FOR
                                     RETRIEVE FUNCTION
NEWDATA  DEFCON C'CHG01'             NEW DATA
KEYAREA  DEFLD  5,0,4                KEY AREA IN DATA AREA
DATAFLD  DEFLD  5,,5                 DATA FIELD AREA
         •
         •
         MVDI   SCPFC,SCPRTUR        SET FOR RETRIEVE
                                     FOR UPDATE
         MVDI   SCPSPID,X'01'        SPA ID
         MVDI   SCPTYP,SCPKEYM       SET FOR KEY ADDRESSING
         MVDI   SCPKEYL,AL1(4)       SET KEY LENGTH
         MVFXD  KEYAREA,DESKFY       SET DESIRED ELEMENT KEY
         MVFXD  SCPDATPT,DATADR      POINT TO KEY AND DATA AREA
LOOP     SCRPAD SCPSTR               EXECUTE THE SCRPAD
                                     INSTRUCTION
         BRAN   OK,UPDATE            IF RETRIEVED GO UPDATE
         LIFOFF SCPRC,SCPRC08,       CHECK IF IN UPDATE STATE
                ERROR
         PAUSE                       WAIT
         JUMP   LOOP                 TRY AGAIN
ERROR      EQUATE                    *
         •
     ERROR ANALYSIS
         •
UPDATE   EQUATE *
         MVDI   SCPFC,SCPRPLR        SET FOR REPLACE FUNCTION
         MVFXD  DATAFLD,NEWDATA      SET NEW DATA
         MVDI   SCPTYP,SCPELMA       SET TO ELEMENT NUMBER
                                     ADDRESSING
         SCRPAD SCPSTR               EXECUTE UPDATE
         BRAN   OK,CONT              CHECK IF SUCCESSFUL
         •
     ERROR ANALYSIS
         •
CONT     EQUATE *
```

Element Number		
1	1234ELM01	
	•	
	•	
23	3678CHG01	Replaced element
24	1289ELM24	
25	unassigned	
	unused 6 bytes	

Add or Retrieve for Update: The SCRPAD instruction with the DEFSCP parameter list can be used to add a new element by key only if an element with that key does *not* already exist in the SPA. If the duplicate keyed element exists, it is retrieved. This function works only for keyed elements. Set the fields in the DEFSCP list as follows:

SCPFC X'25' indicates add/retrieve.

SCPSPID Set to the SPA ID.

SCPTYP X'00', only keyed request allowed.

SCPDATPT Contains the segment number and displacement of a data area containing the element to be added. The beginning characters in the data area are used as a key.

If the element already exists, it is retrieved and placed in this data area.

SCPKEYL Contains the number of element bytes that constitute the key.

When this operation completes, either:

1. The new element in the data area is added to the first available element space of the SPA; the return code is set to X'00'; field SCPELMN contains the element number of the added element.

2. The element exists in the SPA, and is copied to the data area; the return code is set to X'01'; field SCPELMN contains the element number of the retrieved element.

If an error occurs, field SCPRC contains a return code to indicate the status of the operation.

Add or Replace SPA Element: Use the SCRPAD instruction with the COPY DEFSCP parameter list to add or replace a keyed element in the SPA. The DEFSCP parameter list specifies the type of SCRPAD operation, the SPA ID, the address of the new or replacement SPA element, the element key, and its length.

If SCRPAD finds an element with the same key as that in the SCPDATPT data area, the data area contents replace the old SPA element. If none of the SPA elements have the data area key, SCRPAD adds the new element to the SPA. SCRPAD sets a return code, indicating the action taken, in the SCPRC field of the parameter list. Set the COPY DEFSCP parameter list fields as follows:

SCPFC X'23' indicates add or replace an element.

SCPSPID Set to the SPA ID.

SCPTYP X'00' only keyed elements can be added or replaced.

SCPDATPT Specifies the segment and displacement of the new or replacement element. The beginning characters contain the key, as defined by the SCPKEYL field.

SCPKEYL Defines the number of beginning SCPDATPT characters containing an element key.

Note: If the element to be replaced is in update-pending state, only the station that originally retrieved the element can replace it.

Delete a SPA Element: The SCRPAD instruction and the DEFSCP parameter list are used to delete an element from the SPA. The element is located either by element number or by key. The element is then removed from the SPA and its space is made available. Note that elements in "update-pending" state cannot be deleted. To delete an element, the following fields in DEFSCP are completed:

SCPFC X'06' indicates deletion.

SCPSPID Set the SPA ID.

SCPTYP X'00' indicates key addressing;
X'01' indicates an element number will be used.

SCPDATPT Contains the segment number and displacement of a data area containing the key of an element to be deleted. This field is used only for keyed addressing.

SCPKEYL Contains the number of bytes in the data area to be used for the key.

SCPELMN Set to contain the element number of an element to be deleted. This field is used only if the element is located by element number.

On completion, SCRPAD sets a return code in SCPRC to indicate the status of the operation.

For example, if SPA 01 contains 25 elements, and element 1 is to be deleted, the code might look like this:

```
        •
        MVDI     SCPFC,SCPDLER SET FOR DELETE FUNCTION
        MVDI     SCPTYP,SCPELMA SET FOR ELEMENT NUMBER
                 ADDRESSING
        MVDI     SCPSPID,X'01'   SET SPA ID
        MVDI     SCPELMN,AL1(1) SET ELEMENT NUMBER
        SCRPAD     SCPSTR        EXECUTE SCRATCH PAD INSTRUCTION
        BRAN     OK,CONT         CHECK IF SUCCESSFUL
        •
     ERROR ANALYSIS
        •
CONT    EQUATE    *
        •
```

Element Number	
1	available
2	1268ELM02
	•
	•
24	3678ELM24
25	1289ELM25
	unused 6 bytes

Deleting all Elements: The SCRPAD instruction and the DEFSCP parameter list are used to delete all elements in the SPA. Even elements in "update-pending" state are deleted. Set field SCPFC to X'07', and issue SCRPAD. The return code in field SCPRC indicates the status of the operation.

SECTION--Section Control

The SECTION instruction is used to control the active application control section.

Name	Operation	Operand
[label]	SECTION	{ DUMMY / END / INSTR / CONST / AUTO }

DUMMY
Specifies that following instructions are to be placed in a DSECT.

END
Specifies the end of the DSECT.

INSTR/CONST/AUTO
These operands are for 3600 compatibility of split application programs. If your program is 'split', see Appendix F.

Programming Notes

- If a BEGIN, OVLYSEC, or SEGCODE occurs after a SECTION DUMMY statement and before a SECTION END statement, no CSECTs or ADDMEM commands will be generated. See the Host Support User's Guide for further information.
- Each subsequent SECTION specification ends the previous specification except for the dummy section, which is ended only by a SECTION END statement.
- SECTION DUMMY indicates the start of a dummy section (DSECT). The dummy section is used to maintain addressability for the instructions being assembled. Instructions and constants in a dummy section will not be included in the object module. SECTION AUTO, CONST, and INSTR instructions that appear within a dummy section will be ignored.
- SECTION END indicates the end of a dummy section.

For example, the following might be done:

Assembly A:

```
        APOPT     RELOC=Y
ROOT2   BEGIN
ADEF    DEFCON                 1
```

```
BDEF     DEFCON              1
.
.
.
         FINISH
```

Assembly B:

```
         APOPT    RELOC=Y
SECTC    SEGCODE
         SECTION DUMMY
         BEGIN
ADEF     DEFCON              2
BDEF     DEFCON              2
         SECTION  END
         LDLFD    R3,ADEF
         ENDSEG
```

Notes:

1. In assembly A, constants would be generated for the DEFCONs.
2. In assembly B, constants would not be generated but the SECTION DUMMY would allow the assembler to resolve the reference to ADEF.

We recommend that you define a library member that consists of the constant definitions contained in the root section. This library member should include the BEGIN statement and may include EQUATE and DEFLD instructions. When assembling the root section, you should code a COPY statement as the first instruction to copy the library member into the source code. When assembling an application section separate from the root section, you should code a SECTION DUMMY statement followed by a COPY statement to copy the root constants into the source code and then close the dummy section by coding a SECTION END statement. You should follow the dummy section with the application code.

SEGALLOC--Segment Allocate

The SEGALLOC allows you to dynamically allocate segments from a pool defined during configuration using the TRANPL macro or from the general storage pool. Dynamic allocation provides more efficient use of controller storage by allowing run-time management of segment storage. The application issuing the SEGALLOC may request that control not be returned to it until sufficient space is available to satisfy the request. The 4700 places the station in a wait state until the space becomes available. At that time, the request is honored and the station is taken out of the wait state. After it is obtained, the segment space remains allocated until the program that originally allocated it (via SEGALLOC), releases it - explicitly via SEGFREE or implicitly via APRETURN.

Your program addresses a dynamically-allocated segment the same as it addresses any other segment.

Only segments 2-12 may be allocated or freed dynamically and there is no sharing of segments between stations.

You can request up to 64 simultaneous allocations for a work station.

Name	Operation	Operand
[label]	SEGALLOC	{ defld2 | (defrf2) | (reg2) | seg2,disp2 } [,WAIT={Y|N}]

operand 2
Is a field containing the parameter list. The format of this parameter list is:

Byte 0 = one-byte function code - hex 01 = allocate storage

Byte 1 = one-byte segment number

bits 0-3 = zeros
bits 4-7 = segment number

Bytes 2-3 = two-byte length ranging from 1 to 65 535

Byte 4 = one-byte segment-class pool ID from which the segment is to be allocated

WAIT
Y means that the station will be placed in the wait state if the storage is not available. WAIT=N means that a condition code will be set and operation will continue with the next sequential instruction.

Condition Codes: One of the following is set.

Hex Code	Possible Mnemonic	Explanation
01	OK	Successful execution.
04		No storage available.

Program Checks (hex): 01, 02, 09, 11, 24, or 25 can be set.

Hex Code	Explanation
01	The specified parameter list is in an undefined segment or the parameter list specifies an in-use segment number.
02	The parameter list length extends beyond the segment.
09	Invalid operation code.
11	The parameter list specifies an invalid segment number.
25	No room in the segment header area.
29	Size requested in parameter list is larger than the largest pool area defined.

Programming Notes: The program that issues the SEGALLOC instruction and the WAIT=Y operand places the work station for which it is operating, in the wait state until an area becomes available.

SEGCODE--Application Program Section Identifier

The section following the SEGCODE instruction is to be added to a root or overlay section. SEGCODE provides for separately assembled sections that will be connected by the Host Support link-editing function. For a section that is not a root or overlay, the first instruction must be SEGCODE.

The SEGCODE generates an ADDMEM card and a CSECT with the name specified in SEGCODE.

Name	Operation	Operand
[label]	SEGCODE	NAME=name [,VERSION= {version | 1}] [,INSNAME=name]

(1 is underlined as the default.)

name
: Is a 1 to 8 character name identifying the section.

version
: Is the version number (1 to 255) of the section to be stored.

INSNAME
: Applies to split application programs. See Appendix F.

SEGCOPY--Segment Copy

SEGCOPY copies the contents of a field in one segment into a field in a segment associated with the same or a different logical work station. The copy can be either to or from a field in storage belonging to the station currently in control. The beginning of the field that is to contain the copy is indicated by the primary field pointer. A SEGCOPY operation cannot be done to or from station 1 (the System Monitor).

SEGCOPY uses a 6-byte parameter list to refer to the other station and field involved in the data transfer.

The SEGCOPY parameter list defines the second segment and field that either receives or is a source of data transferred by SEGCOPY. If SEGCOPY specifies TO, this field is copied into the *operand 1* segment field. If FROM is specified, the second segment field defined by the parameter list receives a copy of the *seg1*.

Byte 0 = Station ID
Is the binary station ID of the station that owns the second segment involved in the data transfer.

Byte 1 = Segment/Section Indicator
Is a character that further identifies the second segment involved in the data transfer. This field is ignored unless the second segment number is 0 or 14.

If the second segment number is 0, this field can be specified as a character A or B, to identify which Segment 0 of a shared station is being referenced. Unless a value of B is specified, A is assumed.

Byte 2 = Segment Space ID and Number
Bits 0 - 3 contain the segment space ID, Bits 4 - 7 contain the segment number.

Segment space IDs can range from 1 to 15. A value of 0 indicates the current segment space ID of the station specified in the first byte of the parameter list.

If field 2 is to receive a copy of field 1 (the FROM operand is specified), the segment number cannot be 14.

Note: If Segment 0 and B is specified in the parameter list and no Segment 0 or B exists, a condition code of hex 04 is set, and no copying is done.

Bits 4-7 are the binary segment number of field 2. If field 2 is to receive a copy of field 1 (the FROM operand is specified), the segment number cannot be 14.

Byte 3 = Length
Is the binary length of field 2.

Bytes 4 and 5 = Displacement

Is a 2-byte binary number indicating the location of field within the specified segment.

Name	Operation	Operand
[label]	SEGCOPY	seg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2 } , { TO | FROM }

operand 1

Is a field in the specified segment, associated with the station currently in control, that contains field 1. If field 1 is to receive a copy of field 2 (TO is specified). Field 1 must not be in Segment 14.

operand 2

Is a field containing the parameter list. The length associated with this operand is ignored, and the first 6 bytes are assumed to be the parameter list.

When you use register addressing to locate a parameter list, the parameter list can be located in a noncurrent segment space. However, if the parameter list contains the address (segment,displacement) of other storage areas (that is, input/output areas, tables), the other storage areas are always in the current segment space.

TO

Indicates that data is copied from field 2 to operand 1.

FROM

Indicates that data is copied from operand 1 to field 2.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	The data was copied successfully.
02	ID	An invalid station ID was specified.
04	IS	An invalid segment number or segment space ID was specified in the parameter list; or Segment 0 and B were specified in the parameter list, but no Segment 0 for operator B exists (this condition causes an error to occur).
08	OV	There was insufficient room in operand 1 or field 2.

Program Checks (hex): 01, 02, or 27 can be set. Program check 02 is set if the parameter list used by SEGCOPY starts less than 6 bytes from the end of the operand 2 segment.

Programming Notes: The following shows how to transfer data from the processing station to station 2. Note that SEGCOPY does not cause the other station to gain control, but merely transfers data.

```
STALOC      DEFCON     X'02C100400060'     1
              •
TRANSMIT    SETFPL     0,96,64             2
            SEGCOPY    0,STALOC,FROM       3
            BRAN       X'0E',ERRMSG3       4
```

1 Defines the parameter list referred to by SEGCOPY. The list specifies station 2, operator A, Segment 0 of the current segment space id, a length of 64 bytes (X'40') and a displacement into Segment 0 of 96 bytes (X'0060'), immediately following the registers.
2 Sets the PFP of Segment 0 to 96.
3 Transfers 64 bytes of data from this Segment 0 to the Segment 0 associated with station 2, operator A.
4 Branches to an error routine if the transfer is unsuccessful.

SEGFREE--Segment Free

SEGFREE returns segment space allocated using the SEGALLOC instruction to a storage pool.

Your program can free Segments 2 through 12. You can also free segment space by performing an APRETURN.

This instruction requires the TRANPL macro in your system configuration.

Name	Operation	Operand
[label]	SEGFREE	{ defld2 / (defrf2) / (reg2) / seg2,disp2 }

operand 2

Is a field containing a two-byte parameter list. The format of the parameter list is:

byte 0 = function code - hex 01 = release storage

byte 1 = one-byte segment number

bits 0-3 = zeros
bits 4-7 = segment number

Condition Codes: One of the following is set:

Hex Code	Explanation
01	Successful operation.
02	Segment not defined by SEGALLOC or invalid segment for this segment space ID or invalid segment space ID.

Program Checks (hex): 01, 02, or 11 can be set.

Programming Notes: Segments allocated in a segment space may be freed in that space only. For example, program A allocates a segment using SEGALLOC and calls (APCALL) program B. Program B cannot free the segment allocated by program A. The segment allocated by program A is automatically freed if program A issues an APRETURN.

SELECT--Select Segment 0

SELECT allows the controller application program to select which Segment 0 is to be associated with the shared logical work station currently in control. Before the instruction is executed, the character *A* or *B* must be stored in SMSABK.

Name	Operation	Operand
[label]	SELECT	

Condition Codes: One or more of the following are set:

Hex Code	Possible Mnemonic	Explanation
01		Segment 0 is unchanged.
02		Segment 0 is changed.
04	IO	A or B was not specified; A has been set by default.
08	NO	Segment (0) (B) does not exist; Segment 0 (A) remains in control.

Program Checks: None are set.

Programming Notes: The following is an example of the use of the SELECT instruction.

```
TRANS0      DEFCON C'A'                    1
ADD0        DEFCON C'B'                    2

SEGSEL      MVFXD  SMSABK,ADD0             3
            SELECT                         4

                                           5

            MVFXD  SMSABK,TRANS0           6
            SELECT
```

1 Defines the character that indicates which Segment 0 is used for normal transaction processing.
2 Defines the character that indicates which Segment 0 is used for the adding-machine function.
3 Moves the select character (B) to Segment 1.
4 Selects the "B" Segment 0.
5 Performs the adding-machine functions.
6 Selects the Segment 0 used for normal processing before exiting.

SETFLDI--Set Field Immediate

This instruction propagates a byte of data (operand 2) through the field defined in operand 1.

Name	Operation	Operand
[label]	SETFLDI	{ defld1 / (defrf1) / (reg1) / seg1,disp1,len1 } , immdata2

operand 1

Is a field through which the operand 2 data is propagated. The field must not be in Segment 14. The length of operand 1 can be 0 through 65 534 bytes.

operand 2

Is a one-byte value. This byte of data is propagated through the field defined by operand 1.

Condition Codes: The code is not changed by this instruction.

Program Checks (hex): 01, 02, 03, or 27 can be set.

SETFPL--Set Primary Field Pointer and Field Length Indicator

SETFPL modifies the primary field pointer (PFP) and the field length indicator (FLI) of the segment referred to in the instruction. The PFP can be modified arithmetically or logically:

Arithmetic: The PFP can be set to an absolute location within the segment, or to a positive or negative displacement from its current position.

Logical: The PFP can be altered based on delimiter characters that appear within the data in the segment.

The second operand of the SETFPL instruction is used to set the PFP; the third operand is used to set the FLI. At least one of these operands must be specified in the SETFPL instruction.

Any change to the PFP sets the FLI to the difference between the new position of the PFP and the position of the next delimiter found in a scan toward the end of the segment (this includes cases where the FLI operand is not specified) or the difference between the new position of the PFP and the end of the segment if no delimiters are found. If both the PFP and FLI operands are specified, and the FLI operand is a signed number, the FLI is further modified as indicated by the operand; if the FLI operand is an absolute number, the FLI is set to that value.

Delimiters are defined using the DEFDEL instruction. Depending on the specification of the DEL operand of the BEGIN instruction, there may be one or multiple delimiter tables being used.

If [±]*n*, (reg1), or S*(reg1)* is specified for the second operand and the third operand is not specified, +0 is used as the third operand; thus, when the PFP is modified, an inverse modification is performed on the FLI. The FLI is changed only in relation to the new value of the PFP and is not modified further. Similarly, if the third operand is specified but the second is not, +0 is used for the second operand; thus, there is no change to the PFP. If [±] F or L is coded for the second operand and no third operand is specified, no value is assumed for the third operand, and a 2-byte machine instruction is generated. If [±] *n* is coded, a 6-byte machine instruction is generated when +n is greater than +127, -n is less than -128, or n is greater than 255.

Name	Operation	Operand
[label]	SETFPL	{ defld seg, { [±]F | L } seg, { [±]F | L } , { [±]n | (reg2) | S(reg2) } seg, [[±]n | (reg1) | S(reg1) | +0] [, { [±]n | (reg2) | S(reg2) | +0 }] }

defld

Is the label of the DEFLD or DEFCON instruction whose definition (segment, location, and length) is to be used as the new values for the PFP and FLI.

Note: The label of a relocatable DEFCON should not be used.

seg

Is the number of the segment to be modified.

To change the PFP (second operand):

[±]F

Indicates that the PFP is to be modified using delimiters within the data in the segment:

-F

The PFP is set to the first byte to the right of the second delimiter encountered in scanning toward the beginning of the segment (that is, at the start of the preceding field). If the PFP currently points to the first field in the segment, the PFP is set to the beginning of the current field or the beginning of the segment. Unless an FLI value is specified, the FLI is set to the length of the new field.

+F

The PFP is set to the first byte to the right of the next delimiter encountered in scanning toward the end of the segment. If the PFP currently points to the last field in the segment, the PFP is set to the last byte of that segment. Unless an FLI value is specified, the FLI is set to the length of the new field.

F

The PFP is set to the first byte to the right of the first delimiter encountered in scanning toward the beginning of the segment (that is, at the start of the field it currently points to). Unless an FLI value is specified, the FLI is set to the length of the current field.

L

The PFP is increased (that is, moved toward the end of the segment) by an amount equal to the length indicated by the current FLI. Unless an FLI value is specified, the FLI is set to the difference between the new PFP value and the location of the next delimiter encountered in scanning toward the end of the segment.

[±]n

Is a signed or absolute number that indicates the movement or new location of the PFP and a change to the FLI (if the FLI operand is also a signed or absolute number, however, that operand further changes the FLI; refer to the descriptions of the third operand):

-n

Is a negative decimal number (from -0 to -32 768) that specifies the number of bytes the PFP is to be decreased (PFP movement *n* characters toward the beginning of the segment). The FLI is increased by the same value.

+n

Is a positive decimal number (from +0 to +32 767) that specifies the number of bytes the PFP is to be increased (PFP movement *n* characters toward the end of the segment). The FLI is decreased by the same value.

n

Is an absolute decimal number (from 0 to 65 535) that specifies a new value (absolute displacement) for the PFP. The FLI is set to the difference between the new value of the PFP and the position of the first delimiter encountered in a scan toward the end of the segment. If *n* is 0, the PFP points to the first byte of the segment, and the FLI is set to the length of the first field in the segment.

(reg1)

Is the number of a register (0 to 15), in parentheses, that contains a binary value for setting the PFP. The value must be in the rightmost 2 bytes of the register and may range from 0 to 65 535.

S(reg1)

Is the number of the register (0 to 15), preceded by an S and in parentheses, that contains a signed binary value to be algebraically added to the PFP. The value must be in the rightmost 2 bytes. If the leftmost bit of this value is 0, the value is positive and between 0 and 32 767; if the leftmost bit is 1, the value is negative and between 0 and 32 768.

To change the FLI (third operand):

[±]n

Is a signed or absolute number for modifying the FLI:

-n

Is a negative decimal number from -1 to -32 768 that is used to decrease the FLI.

+n

Is a positive decimal number from +0 to +32 767 that is used to increase the FLI.

n

Is a decimal number from 0 to 65 535 that replaces the value of the FLI.

(reg2)

Is the number of a register (0 to 15), in parentheses, that contains a binary value for setting the FLI. The value must be in the rightmost 2 bytes of the register and may range from 0 to 65 535.

S(reg2)

Is the number of the register (0 to 15), preceded by an S and in parentheses, that contains a signed binary value to be algebraically added to the FLI. The value must be in the rightmost 2 bytes. If the leftmost bit of the value is 0, the value is positive and between 0 and 32 767; if the leftmost bit is 1, the value is negative and between 0 and 32 768.

Notes:

1. The PFP and FLI never become negative. If the specified value would cause the PFP to become negative, the PFP is set to 0. If the specified value would cause the FLI to become negative, program check 3 occurs.

2. Program check 0D is set if a SETFPL instruction requiring a delimiter table is issued, and no delimiter table has been defined for the controller application program.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, 03, or 0D may be set.

4-Byte Instruction

Format		Operation		Generated Machine Instruction			
		FP	FLI	OP Code Hex	Byte 2 Binary	Byte 3 Binary	Byte 4 Binary
SETFPL	seg,F	<+1	>+0	14	ssss u000		
SETFPL	seg,+F	>+1	>+0	14	ssss u001		
SETFPL	seg,-F	<<+1	>+0	14	ssss u010		
SETFPL	seg,L	+FLI	>+0	14	ssss u100		
SETFPL	seg,F,N	<+1	N	18	ssss 0u10	uuuu u000	NNNN NNNN
SETFPL	seg,F,+N	<+1	>+N	18	ssss 0u11	uuuu u000	+NNN NNNN
SETFPL	seg,+F,N	>+1	N	18	ssss 0u10	uuuu u001	NNNN NNNN
SETFPL	seg,+F,+N	>+1	>+N	18	ssss 0u11	uuuu u001	+NNN NNNN
SETFPL	seg,-F,N	<<+1	N	18	ssss 0u10	uuuu u010	NNNN NNNN
SETFPL	seg,-F,+N	<<+1	>+N	18	ssss 0u11	uuuu u010	+NNN NNNN
SETFPL	seg,L,+N	+FLI	>+N	18	ssss 0u11	uuuu u100	+NNN NNNN
SETFPL	seg,n,+N	n	>+N	18	ssss 1011	nnnn nnnn	+NNN NNNN
SETFPL	seg,L,N	+FLI	N	18	ssss 0u10	uuuu u100	NNNN NNNN
SETFPL	seg,n,N	n	N	18	ssss 1010	nnnn nnnn	NNNN NNNN
SETFPL	seg,+n,N	+n	N	18	ssss 1110	+nnn nnnn	NNNN NNNN
SETFPL	seg,+n,+N	+n	-n+N	18	ssss 1111	+nnn nnnn	+NNN NNNN
SETFPL	seg,F,(reg)	<+1	(R)	19	ssss 0u10	uuuu u000	uuuu RRRR
SETFPL	seg,F,S(reg)	<+1	>+(R)	19	ssss 0u11	uuuu u000	uuuu RRRR
SETFPL	seg,+F,(reg)	>+1	(R)	19	ssss 0u10	uuuu u001	uuuu RRRR
SETFPL	seg,+F,S(reg)	>+1	>+(R)	19	ssss 0u11	uuuu u001	uuuu RRRR
SETFPL	seg,-F,(reg)	<<+1	(R)	19	ssss 0u10	uuuu u010	uuuu RRRR
SETFPL	seg,-F,S(reg)	<<+1	>+(R)	19	ssss 0u11	uuuu u010	uuuu RRRR
SETFPL	seg,L,S(reg)	+FLI	>+(R)	19	ssss 0u11	uuuu u100	uuuu RRRR
SETFPL	seg,n,S(reg)	n	>+(R)	19	ssss 1011	nnnn nnnn	uuuu RRRR
SETFPL	seg,L,(reg)	+FLI	(R)	19	ssss 0u10	uuuu u100	uuuu RRRR
SETFPL	seg,n,(reg)	n	(R)	19	ssss 1010	nnnn nnnn	uuuu RRRR
SETFPL	seg,+n,(reg)	+n	(R)	19	ssss 1110	+nnn nnnn	uuuu RRRR
SETFPL	seg,+n,S(reg)	+n	-n+(R)	19	ssss 1111	+nnn nnnn	uuuu RRRR
SETFPL	seg,(reg),+N	(r)	>+N	1A	ssss 1011	uuuu rrrr	+NNN NNNN
SETFPL	seg,(reg),N	(r)	N	1A	ssss 1010	uuuu rrrr	NNNN NNNN
SETFPL	seg,S(reg),N	+(r)	N	1A	ssss 1110	uuuu rrrr	NNNN NNNN
SETFPL	seg,S(reg),+N	+(r)	-(r)+N	1A	ssss 1111	uuuu rrrr	+NNN NNNN
SETFPL	seg,(reg),S(reg)	(r)	>+(R)	1B	ssss 1011	uuuu rrrr	uuuu RRRR
SETFPL	seg,(reg),(reg)	(r)	(R)	1B	ssss 1010	uuuu rrrr	uuuu RRRR
SETFPL	seg,S(reg),(reg)	+(r)	(R)	1B	ssss 1110	uuuu rrrr	uuuu RRRR
SETFPL	seg,S(reg),S(reg)	+(r)	-(r)+(R)	1B	ssss 1111	uuuu rrrr	uuuu RRRR

n = numeric value in binary for Field Pointer
R = register number in binary for Field Length
N = numeric value in binary for Field Length
< = scan backwards to the first delimiter encountered†
> = for FP, scan forwards to the first delimiter encountered†
> = for FLI, the number of consecutive nondelimiter characters beginning with the character located by the FP and scanning forward
<< = scan backwards to the second delimiter encountered†
+(r), -(r), +(R) = specified register contains a signed binary number ranging between +32,767 and -32,768
(r), (R) = specified register contains an absolute number ranging between 0 and 65,535

s = segment number in binary
u = unused bit
r = register number in binary for Field Pointer

† If the scan results in the start or end of the segment, +1 is not added to the field pointer.

Figure 5-3 (Part 1 of 2). Set Field Pointer Instructions Summary

6-Byte Instruction

Format	Operation: FP	Operation: FLI	OP Code Hex	Byte 2 Binary	Byte 3 Binary	Byte 4 Binary	Byte 5 Binary	Byte 6 Binary
SETFPL seg,F,N	<+1	N	5F	ssss 0000	uuuu 0000	uuuu uuuu	NNNN NNNN	NNNN NNNN
SETFPL seg,F,+N	<+1	>+N	5F	ssss 0001	uuuu 0000	uuuu uuuu	+NNN NNNN	NNNN NNNN
SETFPL seg,+F,N	>+1	N	5F	ssss 0000	uuuu 0001	uuuu uuuu	NNNN NNNN	NNNN NNNN
SETFPL seg,+F,+N	>+1	>+N	5F	ssss 0001	uuuu 0001	uuuu uuuu	+NNN NNNN	NNNN NNNN
SETFPL seg,-F,N	<<+1	N	5F	ssss 0000	uuuu 0010	uuuu uuuu	NNNN NNNN	NNNN NNNN
SETFPL seg,-F,+N	<<+1	>+N	5F	ssss 0001	uuuu 0010	uuuu uuuu	+NNN NNNN	NNNN NNNN
SETFPL seg,L,N	+FLI	N	5F	ssss 0000	uuuu 0100	uuuu uuuu	NNNN NNNN	NNNN NNNN
SETFPL seg,L,+N	+FLI	>+N	5F	ssss 0001	uuuu 0100	uuuu uuuu	+NNN NNNN	NNNN NNNN
SETFPL seg,n,N	n	N	5F	ssss 1000	nnnn nnnn	nnnn nnnn	NNNN NNNN	NNNN NNNN
SETFPL seg,n,+N	n	>+N	5F	ssss 1001	nnnn nnnn	nnnn nnnn	+NNN NNNN	NNNN NNNN
SETFPL seg,n,(reg)	n	(R)	5F	ssss 1010	nnnn nnnn	nnnn nnnn	uuuu uuuu	uuuu RRRR
SETFPL seg,n,S(reg)	n	>+(R)	5F	ssss 1011	nnnn nnnn	nnnn nnnn	uuuu uuuu	uuuu RRRR
SETFPL seg,+n,N	+n	N	5F	ssss 1100	+nnn nnnn	nnnn nnnn	NNNN NNNN	NNNN NNNN
SETFPL seg,+n,+N	+n	-n+N	5F	ssss 1101	+nnn nnnn	nnnn nnnn	+NNN NNNN	NNNN NNNN
SETFPL seg,+n,(reg)	+n	(R)	5F	ssss 1110	+nnn nnnn	nnnn nnnn	uuuu uuuu	uuuu RRRR
SETFPL seg,+n,S(reg)	+n	-n+(R)	5F	ssss 1111	+nnn nnnn	nnnn nnnn	uuuu uuuu	uuuu RRRR
SETFPL seg,(reg),N	(r)	N	5F	ssss 0000	uuuu 1000	uuuu rrrr	NNNN NNNN	NNNN NNNN
SETFPL seg,(reg),+N	(r)	>+N	5F	ssss 0001	uuuu 1000	uuuu rrrr	+NNN NNNN	NNNN NNNN
SETFPL seg,S(reg),N	+(r)	N	5F	ssss 0100	uuuu 1000	uuuu rrrr	NNNN NNNN	NNNN NNNN
SETFPL seg,S(reg),+N	+(r)	-(r)+N	5F	ssss 0101	uuuu 1000	uuuu rrrr	+NNN NNNN	NNNN NNNN

n = numeric value in binary for Field Pointer
R = register number in binary for Field Length
N = numeric value in binary for Field Length
< = scan backwards to the first delimiter encountered†
> = for FP, scan forwards to the first delimiter encountered†
> = for FLI, the number of consecutive nondelimiter characters beginning with the character located by the FP and scanning forward
<< = scan backwards to the second delimiter encountered†
+(r), -(r), +(R) = specified register contains a signed binary number ranging between +32,767 and -32,768
(r), (R) = specified register contains an absolute number ranging between 0 and 65,535

s = segment number in binary
u = unused bit
r = register number in binary for Field Pointer

† If the scan results in the start or end of the segment, +1 is not added to the field pointer.

Figure 5-3 (Part 2 of 2). Set Field Pointer Instructions Summary

SETSFP--Set Secondary Field Pointer

SETSFP modifies the secondary field pointer (SFP). If *seg* is specified and the second operand is not, the SFP is set to the value of the primary field pointer (PFP).

The SFP can be modified arithmetically or logically:

Arithmetic: The SFP can be set to an absolute location within a segment, or to a positive or negative displacement from its current position.

Logical: The SFP can be altered based on delimiter characters that appear within the data in the segment.

Delimiters are defined using the DEFDEL instruction. Depending on the specification of the DEL operand of the BEGIN instruction, there may be one or multiple delimiter tables being used.

Name	Operation	Operand
[label]	SETSFP	{ defcon | defld | seg [, { [±]n | (reg1) | S(reg1) | [±]F | L }] }

defld

Is the label of a DEFLD or DEFCON whose definition (segment and location) is to be used for setting the SFP. The displacement of the DEFLD or DEFCON specified must not be greater than 255. The label of a relocatable DEFCON should not be used.

seg

Is the number of the segment whose SFP is to be changed.

[±]n

Is a signed or absolute number that indicates the new location of the SFP.

-n

Is a negative decimal number (from -0 to -128) that specifies the number of bytes the SFP is to be decreased.

+n

Is a positive decimal number (from +0 to +127) that specifies the number of bytes the SFP is to be increased.

n

Is an absolute decimal number (from 0 to 255) that specifies a new value for the SFP.

(reg1)

Is the number of the register (0 to 15), in parentheses, that contains a binary value for setting the SFP. The value must be in the rightmost 2 bytes of the register and may range from 0 to 65 535.

S(reg1)

Is the number of a register (0 to 15), preceded by an S and in parentheses, that contains a signed binary value to be algebraically added to the SFP. The value must be in the rightmost 2 bytes of the register. If the leftmost bit of the value is 0, the value is positive and between 0 and 32 767; if the leftmost bit is 1, the value is negative and between 0 and 32 768.

[±]F

The SFP is modified using delimiters within the data in the segment;

+F

The SFP is set to the first byte after the next delimiter encountered in scanning toward the end of the segment. If the SFP currently points to the last field in the segment, the SFP is set to the last byte of that segment.

-F

The SFP is set to the first byte to the right of the second delimiter encountered in scanning toward the beginning of the segment (that is, at the start of the preceding field). If the SFP currently points to the first field in the segment, the SFP is set to the beginning of the current field or the beginning of the segment.

F

The SFP is set to the first byte after the first delimiter encountered in scanning toward the beginning of the segment (that is, at the start of the field it currently points to).

L

The SFP is increased by an amount equal to the length indicated by the current field-length indicator.

Note: The SFP never becomes negative; if the specified value would create a negative number, the SFP is set to zero.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, or 0D may be set.

Note: Program check 0D is set if a SETSFP instruction requiring a delimiter table is issued, and no delimiter table has been defined for the controller application program.

Programming Notes: The following example shows how the SFP can be set for a DEFLD/DEFCON with a displacement greater than 255:

```
F1     DEFLD     4,256,1     1
F1D    EQUATE    (D:F1)      2
       •
       •
       •
       LDDI      R5,F1D  }   3
       LDRA      R5,F1   }   3
       SETSFP    S4,(R5)     4
```

1 Defines the field to be used when setting the SFP.
2 Assigns a label to the displacement of F1.
3 The F1 displacement can be set in the rightmost 2 bytes of register 5 by using either instruction.
4 Sets the SFP to 256.

4-Byte Instruction

Format	Operation		Generated Machine Instruction			
	FP	FLI	OP Code Hex	Byte 2 Binary	Byte 3 Binary	Byte 4 Binary
SETSFP seg,F	<+1		20	ssss 0000	uuuu u000	uuuu uuuu
SETSFP seg,+F	>+1		20	ssss 0000	uuuu u001	uuuu uuuu
SETSFP seg,-F	<<+1		20	ssss 0000	uuuu u010	uuuu uuuu
SETSFP seg,L	+FLI		20	ssss 0000	uuuu u100	uuuu uuuu
SETSFP seg,n	n		20	ssss 1000	nnnn nnnn	uuuu uuuu
SETSFP seg,+n	+n		20	ssss 1100	+nnn nnnn	uuuu uuuu
SETSFP seg,(reg)	(r)		22	ssss 1000	uuuu rrrr	uuuu uuuu
SETSFP seg,S(reg)	+(r)		22	ssss 1100	uuuu rrrr	uuuu uuuu
SETSFP seg	PFP		52	ssss 0000		

n = numeric value in binary for Field Pointer
R = register number in binary for Field Length
N = numeric value in binary for Field Length
s = segment number in binary
u = unused bit
r = register number in binary for Field Pointer
< = scan backwards to the first delimiter encountered†
> = for FP, scan forwards to the first delimiter encountered†
> = for FLI, the number of consecutive nondelimiter characters beginning with the character located by the FP and scanning forward
<< = scan backwards to the second delimiter encountered†
+(r), -(r), +(R) = specified register contains a signed binary number ranging between +32,767 and -32,768
(r), (R) = specified register contains an absolute number ranging between 0 and 65,535

† If the scan results in the start or end of the segment, +1 is not added to the field pointer.

Figure 5-4. Set Field Pointer Instructions Summary

SHIFTL--Shift-Left Data in a Register

SHIFTL shifts the contents of a register a specified number of bit positions to the left. Data that is moved past the leftmost position of the register is lost. Zeros are shifted into the rightmost position of the register as the operation takes place.

Name	Operation	Operand
[label]	SHIFTL	reg1,count

operand 1
Is a register whose contents are to be shifted.

count
Is the decimal number of bit positions (1-16) the contents are to be shifted.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	NL	No significant bits were lost.
08	BL	Significant (one) bits were lost.

Program Checks (hex): None are set.

SHIFTR--Shift-Right Data in a Register

SHIFTR shifts the contents of a register a specified number of bit positions to the right. Data that is moved past the rightmost position of the register is lost. Zeros are shifted into the leftmost position of the register as the operation takes place.

Name	Operation	Operand
[label]	SHIFTR	reg1,count

operand 1
Is a register whose contents are to be shifted.

count
Is the decimal number of bit positions (1-16) the contents are to be shifted.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	NL	No significant bits were lost.
08	BL	Significant (one) bits were lost.

Program Checks: None are set.

SINIT--Start Initialization Section

Use this instruction to begin a section in your application program that initializes segments by work station ID. The values this section defines apply only when this program operates as the initial application for the work station, but not for any programs called by this program. The initialization section must follow the FINISH instruction and contain only INITSEG instructions, but can contain as many as you require. The values specified in the INITSEG instructions replace any values specified for the same fields during configuration (CPGEN) by a SEGINIT macro.

The format of the SINIT instruction is:

Name	Operation	Operand
label	SINIT	

Note: The SINIT instruction has no operands but its label is required.

| STATS--Obtain or Reset Extended Statistical Counters

The STATS instruction provides access to the information maintained in the extended statistical counters that were defined during the configuration process by the EXTCTR macro. It also provides access to the basic statistical counters for a specified device. STATS performs the following functions according to the request code:

Request Code 01 Return data associated with the specified extended statistical counters for ESCID (X'01'). For the specified extended statistical counter ID (ESCID), return the first device ID, the total number of bytes transmitted, the number of data bytes in error, and the number of devices assigned to the ESCID. You can set optional flags in the parameter list requesting that the counter also be reset (RESET flag), or that the counter to be returned is the next one in the table (NEXT flag).

Request Code 02 Return the physical device address associated with the specified extended statistical counter. For the specified ESCID, return the first device ID. You can use the NEXT flag to obtain each of the additional DEVIDs in turn.

Request Code 03 Reset all extended statistical counters.

Request Code 04 Return the extended statistical counter ID for the specified physical device address.

Request Code 05 Return basic statistical counters for the physical device address specified. You can set optional flags in the parameter list requesting that the counter also be reset (RESET flag), or that the counter to be returned is the next one in the table (NEXT flag).

Request Code 06 Reset all basic statistical counters.

The STATS instruction points to a parameter list (see the DEFESP copy file in Appendix B) that contains a request code. The request code defines the function to be performed and any parameters required for that request. The beginning of the parameter list is defined by operand 2. The length of the parameter list is implied by the request code. Generally, if operand 2 refers to a segment that has insufficient space, a program check occurs when the instruction is performed. For a request code of X'05' (return statistical counters for the specified physical device address), a program check occurs if there is insufficient space for at least the first two counters. Otherwise, insufficient space causes a condition code of C'04'. The PFP and the FLI for the segment that contains the parameter list remains unchanged after STATS is executed.

Name	Operation	Operand
[label]	STATS	{ defld2 | (defrf2) | (reg2) }

operand 2
Defines the start of the parameter list. The length associated with the operand is ignored. The parameter is assumed to have one of two formats. For request codes X'01', X'02', X'03', and X'04', format 1 is used as explained in Figure 5-5 on page 5-373. For request codes X'05' and X'06', format 2 is used as explained in Figure 5-6 on page 5-374.

Note: DEFCON label cannot be used. Also, the segment number in the DEFLD instruction cannot be 14.

Condition Codes: One of the following codes is set:

Hex Code	Explanation:
01	The requested function was performed successfully.
02	There was an invalid device or extended counter specification.
04	There was insufficient space in the segment to store all of the device statistical counters.

Program Checks (hex): 01, 02, 09, 11, or 27 can be set.

Program check 01 occurs if the segment to which DEFLD instruction refers is 14.

Program check 02 occurs if there is insufficient space for the requested information.

Program check 09 occurs if the EXTCTR macro was not coded in the CPGEN or if loading of the extended counter module was suppressed by the control operator.

Program check 11 occurs if the request code is not one of those listed in Figure 5-5 on page 5-373 or Figure 5-6 on page 5-374. Program check 27 is set when a register address contained an invalid segment space ID.

Request Code	Flags	ESCID	DEVID	Total Bytes	Error Bytes	Num. of Devices

Byte 0 1 2 4 6 12 16 17

Request Code
Is a 1-byte request code, as follows:

Code	Meaning
X'01'	For the specified ESCID, return the first DEVID, total bytes, error count, and number of devices assigned to an extended counter.
X'02'	For the specified ESCID, return a DEVID associated with the extended counter.
X'03'	Reset all extended counters.
X'04'	For the specified DEVID, return the corresponding ESCID.

Flags
Is an 8-bit modifier as follows:

Flag	Name	Meaning
X'80'	RESET	Reset (zero) the extended counter after read-out (Request Code = 01).
X'40'	NEXT	The statistical counter to be accessed is the counter after the one addressed by the parameter list.

ESCID
Is a 2-byte hexadecimal (X'xxxx') ID specified on the EXTCTR macro.

DEVID
Is a 2-byte physical device address consisting of loop, terminal loop address, component, and subaddress.

Total Bytes
Is the number of bytes transmitted from the input device(s).

Error Count
Is the number of error bytes detected on the input transmission.

Number of Devices
Is the number of devices assigned to this extended counter.

Figure 5-5. Format 1 Request and Information Returned by STATS

Request Code	Flags	DEVID	Device Type	Feature Flags	Station ID	Num. of Counters	Statistical Counters
Byte 0	1	2	4	5	6	7	8 (1 byte each)

Request Code
Is a 1-byte request code, as follows:

Code	Meaning
X'05'	For the specified DEVID, return the basic statistical counters associated with the device.
X'06'	Reset all basic statistical counters.

Flags
Is an 8-bit modifier, as follows:

Flag	Name	Meaning
X'80'	RESET	Reset (zero) the statistical counters associated with the device after read-out.
X'40'	NEXT	The statistical counters to be accessed are the next counter after the one addressed by the parameter list.

DEVID
Is a 2-byte physical device, loop, or port address. The following is the format table of allowable DEVID entries.

Diskette	X'9'	X'0'	X'2' or X'3' *	X'0'
Link	X'9'	X'0'	X'1'	X'0'
Loop	Loop	X'0'	X'0'	X'0'
Loop Device	Loop	Terminal	Component	X'0'
DCA Adapter	X'9'	X'A'	X'0'	X'0'
DCA Port	X'A'	Port	X'0'	X'0'
DCA Device	X'A'	Port	Component	X'0'

* X'2' = primary diskette; X'3' = secondary diskette

Figure 5-6 (Part 1 of 3). Format 2 Request and Information Returned by STATS

Loop

Is the 4-bit binary loop number (1-4) assigned during the controller configuration procedure.

Terminal

Is the 4-bit binary terminal address established at the terminal by setting address switches on the terminal itself.

Port

Is the 4-bit binary number (0-7) of the DCA port.

Component

Is the 4-bit component address of a terminal component, as follows:

Component Address	Component
0001	4704/3604/3278/3279 Keyboard
0010	4704/3604/3278/3279 Display
0011	4704/3604 Magnetic Stripe Encoder
0100	4710/3610/3612 Document Printer
0101	3611/3612 Passbook Printer
0100	3615 Administrative Terminal Printer (non-address shared)
0110	3606/3608 Keyboard/Display
0111	3608 Printer
1000	3614/3624 Terminal
0100	3262/3287 Printer
nnnn*	3615 Administrative Terminal Printer (address shared)
nnnn*	3616 Journal Printer
nnnn*+1	3616 Passbook/Document Printer

*nnnn = the setting of the unit's subaddress switches

Device Type

Is a 1-byte component code, as follows:

Code	Component
X'01'	Communication Link
X'02'	Diskette
X'03'	ALA Line
X'04'	Disk
X'05'	Encryption Facility
X'80'	Loop
X'81'	4704/3604/3278 Keyboard
X'82'	4704/3604/3278 Display Displaywriter Personal Computer
X'83'	4710/3610/3612 Document Printer 3611/3612 Passbook Printer
X'84'	3262/3287 Printer
X'86'	4704/3604 Magnetic Stripe Encoder
X'87'	3614/3624 Consumer Transaction Facility
X'88'	3606/3608 Keyboard/Display
X'89'	3608 Printer
X'8A'	3615 Administrative Terminal Printer
X'92'	3616 Printer
X'95'	Device Cluster Adapter
X'9A'	4710 Printer
X'AB'	Magnetic Stripe Encoder for 4704 Models 2 and 3
X'B0'	4720 Printer

Figure 5-6 (Part 2 of 3). Format 2 Request and Information Returned by STATS

Feature Flags

Is an 8-bit code. The contents of this byte are device dependent, and the feature flags defined are listed in the COPY file DEFESP. All application program references to these flags should be by use of the symbols defined on DEFESP to avoid source changes or improper execution in case flag values must be redefined in subsequent releases.

No feature flags are defined for the communication link, diskette, or disk. The byte for loops indicates the clocking loop and the configured loop speed. For loop-attached devices, feature flags 5 through 8 are common to all devices and features 1 through 4 vary by device.

Devices for which no symbols are listed in DEFESP do not use the feature flags. Flag values for any listed device that are not defined are reserved and should not be assumed to be either 0 or 1.

Station ID

Is the binary ID number of the station associated with the device.

Number of Counters

Is the binary number of statistical counters that follow.

Statistical Counters

Are the counters themselves (see the *4700 Subsystem Operating Procedures:* GC31-2032 for further information).

Figure 5-6 (Part 3 of 3). Format 2 Request and Information Returned by STATS

STFLD--Store Field

STFLD stores the contents of a register into a field. The data is moved without character conversion. If the field to contain the contents of the register is less than 6 bytes long, the data is truncated on the left. If this field is longer than 6 bytes, the leftmost bit of the register is propagated to the left to pad the rest of the field.

Name	Operation	Operand

```
[label] STFLD    reg1, { defld2             }
                       { (defrf2)           }
                       { (reg2)             }
                       { seg2,disp2,len2    }
```

operand 1
Is a register whose contents are to be stored.

operand 2
Is a field that is to contain the contents of the register. The field must not be in Segment 14, and the field length can be 1-15 bytes unless you specify register addressing, which allows a length ranging from 1 to 4095 bytes.

Condition Codes: One or more of the following are set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, 03, or 27 can be set.

STFLDC--Store Field Character

STFLDC stores the decimal EBCDIC equivalent of the signed binary number in a register into a field. If the field is too small, the converted number is truncated on the left. The maximum value for the converted number is 15 characters. Zero characters C'0' complete the field to the left from the most significant character. The sign is not retained in the stored data. If a sign is desired, the programmer must insert a sign character.

Name	Operation	Operand

```
                               ⎧ defld2           ⎫
[label] STFLDC    reg1,  ⎨ (defrf2)         ⎬
                               ⎪ (reg2)           ⎪
                               ⎩ seg2,disp2,len2  ⎭
```

operand 1
Is a register whose contents are to be converted and stored.

operand 2
Is a field to contain the converted number. The field may not be in Segment 14, and the length of this field can be 1-15 bytes unless you specify register addressing which allows a length ranging from 1 to 4095 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, 03, or 27 can be set.

Programming Notes: All operations are performed algebraically on the contents of two registers or a register and a field. The example below shows a use of the STFLDC instruction.

```
OUTFLD  DEFLD   OUTSEG,0,19      1
OUTTYP  DEFLD   OUTSEG,0,2       2
OUTACCT DEFLD   OUTSEG,,7        3
OUTAMNT DEFLD   OUTSEG,,10       4
        •
        •
        •
        ADDREG R01,R02           5
        BRAN   OV,ERR1           6
        STFLDC R01,OUTAMNT       7
        BRANL  NG,MINUSRTN       8
```

1 Defines the entire output field.
2 Defines the transaction type field within OUTFLD.
3 Defines the account number field. Because the displacement is omitted, the field is assumed to begin at the next byte after OUTTYP.
4 Defines the amount field. Again, the displacement operand is omitted.
5 Adds the amount in REG1 and REG2 and stores the result in REG1.
6 Tests for overflow. Any other condition is ignored.
7 Converts the total to EBCDIC decimal characters and stores the rightmost 10 digits in OUTAMNT.
8 Branches to a subroutine that inserts the minus sign if a negative number was stored.

STOVLY--Start Overlay

STOVLY defines a load point for an overlay section. This instruction may be included either in the root section or in an overlay section. The STOVLY instruction generates an ENTRY instruction.

Note: When you are assembling with the nonrelocatable option (RELOC=N in the APOPT instruction), the STOVLY instruction and the OVLYSEC instruction that refers to it must appear in the same assembly.

When a relocatable overlay section is assembled separately from the application root, EXTRN and ENTRY instructions must be used to establish the connection between the STOVLY and the OVLYSEC instructions.

Name	Operation	Operand
[label]	STOVLY	[C / I / A]

(A is underlined as the default.)

The C, I, and A operands are used only for split application programs. See Appendix F for further information.

STSEG--Store Segment

STSEG stores the contents of a register into a segment-header addressed field. If the field to contain the register contents is less than 6 bytes long, the data is truncated to the left. If this field is longer than 6 bytes, the high-order bit of the high-order byte is propagated to the left to fill the entire field.

Name	Operation	Operand
[label]	STSEG	reg1,seg2

operand 1
Is a register whose contents are to be stored.

operand 2
Is a field in the specified segment that is to receive the register's contents. The field must not be in Segment 14. The location of the field within the segment to contain the register contents is determined by the primary field pointer and the length is determined by the field length indicator. The field length can be from 1 to 4095 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, or 03 may be set.

STSEGC--Store Segment Character

STSEGC stores the decimal EBCDIC equivalent of the binary contents of a register in a segment-header addressed field. Zero characters (C'0') pad the field to the left from the most significant digit. If the field is too small, the converted number is truncated to the left. The maximum value for the converted number is 15 characters. The sign is not retained in the stored data. If a sign is desired, the programmer must insert a sign character.

Name	Operation	Operand
[label]	STSEGC	reg1,seg2

operand 1
Is the register whose contents are to be converted and stored.

operand 2
Is a field in the specified segment that is to receive the register's contents. The field must not be in Segment 14. The location of the field within the segment that is to contain the converted number is indicated by the primary field pointer and the length is indicated by the field length indicator. The field length can be from 1 to 4095 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	TR	Truncation occurred.

Program Checks (hex): 01, 02, 03, or 27 can be set.

SUBFLD--Subtract Field

SUBFLD algebraically subtracts the binary contents of a fixed field from the binary contents of a register and places the result in the register.

The field must be between 0 and 6 bytes long. If a length of 0 is specified, a binary 0 is subtracted from the contents of the register. The leftmost bit of the leftmost byte is assumed to be the sign.

Name	Operation	Operand

```
                              ⎧ defcon2           ⎫
                              ⎪ defld2            ⎪
[label] SUBFLD   reg1,        ⎨ (defrf2)          ⎬
                              ⎪ (reg2)            ⎪
                              ⎩ seg2,disp2,len2   ⎭
```

operand 1
Is a register containing the minuend. At the end of the operation, this register contains the result.

operand 2
Is a field containing the value (subtrahend) to subtract from operand 1. The length of the subtrahend is from 0 to 6 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overflow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, or 27 can be set.

SUBFLDL--Subtract Field Logical

SUBFLDL subtracts the contents of a 6-byte field from the contents of a register. If the field length is less than 6 bytes, the field is treated as a 6-byte field by propagating zeros to the left. The binary contents of the field is then subtracted from the binary contents of the register and the result is placed in the register. If a length of 0 is specified, a binary 0 is subtracted from the contents of the register.

Name	Operation	Operand
[label]	SUBFLDL	reg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2,len2 }

```
                           ( defcon2          )
                           ) defld2           (
[label] SUBFLDL   reg1,    ) (defrf2)         (
                           ) (reg2)           (
                           ( seg2,disp2,len2  )
```

operand 1
Is a register containing the minuend. At the end of the operation, this register contains the result.

operand 2
Is a field containing the value (subtrahend) to subtract from operand 1. The length of the subtrahend is from 0 to 6 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overflow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, or 27 can be set.

SUBREG--Subtract Register

SUBREG algebraically subtracts the binary contents of one register from the binary contents of another register and places the result in the register specified in the first operand.

Name	Operation	Operand
[label]	SUBREG	reg1,reg2

operand 1
Is the register that contains the minuend. At the end of the operation, this register contains the result.

operand 2
Is the register that contains the value (subtrahend) to subtract from operand 1.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overflow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks: None are set.

SUBZ--Subtract Zoned Decimal

This instruction subtracts zoned decimal operand 2 from zoned decimal operand 1, and replaces operand 1 with the result. The length of either operand is from 1 to 63 bytes; if either operand is more than 15 bytes long, it must be selected using register addressing.

Note: This is an optional instruction, and requires that module P31 be specified on the OPTMOD configuration macro.

Name	Operation	Operand

```
                     { defld1            }   { defcon2           }
[label] SUBZ         { (defrf1)          } , { defld2            }
                     { (reg1)            }   { (defrf2)          }
                     { seg1,disp1,len1   }   { (reg2)            }
                                             { seg2,disp2,len2   }
```

operand 1
Is a field containing the zoned decimal minuend (value subtracted from), and the location of the result. If this operand length is greater than the result, each unused high-order byte is set to X'FO'. This operand cannot be located in Segment 14.

operand 2
Is a field containing the zoned decimal subtrahend (the value subtracted).

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	ZO	The result is 0.
02	NG	The result is less than 0.
03	LE	The result is less than or equal to 0.
04	PS	The result is greater than 0.
05	GE	The result is greater than or equal to 0.
06	NE	The result is not equal to 0.
08	OV	An overflow occurred.
09		An overflow occurred and the result is 0.
0A		An overflow occurred and the result is less than 0.
0C		An overflow occurred and the result is greater than 0.

Program Checks (hex): 01, 02, 03, 09, or 27 can be set.

TABLE--Define Table for LSEEK/LSEEKP

TABLE defines a table, in Segment 14, composed of variable-length elements that may have branching locations associated with them. All entries in a table occupy the same amount of storage, because entries shorter than the length specified in the LNG operand are padded on the right with blanks.

The generated TABLE may be referenced by the LSEEK and LSEEKP instructions. The SRT operand is used with the LSEEKP instruction to provide a binary search.

A maximum of 200 entries may be specified in a table. Each operand specified in the TABLE instruction cannot exceed 255 characters including commas, parentheses, and quotation marks. Each element in the instruction can be specified in one of two forms:

1. A mixture of data types, enclosed in parentheses with the data types separated by commas

2. A single data type.

To define a table of elements without associated branching locations, you code TABLE like this:

Name	Operation	Operand
[label]	TABLE	element1[,element2...,elementn],LNG=nnn [,SRT=n] where: element is { data (data1,data2...,datan) }

To define a table of elements with associated branching locations, you code TABLE like this:

Name	Operation	Operand
[label]	TABLE	element1[,element2...,elementn],LNG=nnn [,SRT=n] where: element is { (data,addr) (data1,data2...,datan,addr) }

data

Is the information, called a data item, to be included in this element of the table. It may be specified as hexadecimal (X'nn'), character (C'nn'), an A-type address constant (ALn(nn)), fullword (FLn'nn'), or halfword (HLn'nn').

addr

Is an address constant or the label of an instruction. The LSEEK instruction that refers to this table can be coded so that a match between the field referred to in LSEEK and a table element causes a branch to this location (see the description of LSEEK). The label may be an external symbol when RELOC=Y is specified in the APOPT instruction.

LNG

Is a decimal value from 1 to 255, specifying the length of the entries in the table. The length must be equal to or greater than the largest entry. Entries shorter than the length specified are padded on the right with blanks.

SRT

Specifies that the table is to be sorted, and the number of characters on which to sort. Replace *n* with the number of characters at the beginning of each entry that are to be sorted. This sort field is always a subset of the total entry, and always begins at the beginning of the element.

If you code this operand to sort the table, the first data item of each element must meet these requirements:

- The data type must be either C or X.
- All data items must have the same attribute (C or X).
- If several data items are specified for an element, the element is sorted according to the first data item only.

TSTMSK--Test under Mask

TSTMSK tests 1 or 2 bytes of a field under control of a mask. Each bit position in the mask that contains a 1 causes the corresponding bit position in the field to be tested. TSTMSK returns a condition code that signifies the bits tested were all 0's, all 1's, or mixed, or that the bits tested were identical to the mask.

Name	Operation	Operand

```
                      ⎧ defcon1     ⎫   ⎧ defcon2           ⎫
                      ⎪ defld1      ⎪   ⎪ defld2            ⎪
[label] TSTMSK        ⎨ (defrf1)    ⎬ , ⎨ (defrf2)          ⎬
                      ⎪ (reg1)      ⎪   ⎪ (reg2)            ⎪
                      ⎩ seg1,disp1  ⎭   ⎩ seg2,disp2,len2   ⎭
```

operand 1
Is a field to be tested. The number of bytes to be tested is determined by the length of the mask.

operand 2
Is a field containing the mask. The length must be either 1 or 2 bytes.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	MZ	All tested bits are 0's or the mask bits are all 0's.
02	MX	The tested bits are mixed 1's and 0's.
04	ME	The tested field and the mask are identical.
08	MO	All tested bits are 1's.

Note: Condition code hex 04 is set with hex 08; it is never set alone. This may affect the result of a branch (or JUMP) instruction that uses this code as the condition for the branch.

Program Checks (hex): 01, 02, 03, or 27 can be set.

TSTMSKI--Test under Mask Immediate

TSTMSKI tests 1 or 2 bytes of a field under control of a mask made up of 1 or 2 bytes of data. Each bit position in the mask that contains a 1 causes the corresponding bit position in the field to be tested. TSTMSKI returns a condition code that signifies the bits tested were all 0's, all 1's, or mixed, or that the bits tested were identical to the mask.

Name	Operation	Operand
[label]	TSTMSKI	{defcon1 | defld1 | (defrf1) | (reg1) | seg1,disp1},immdata2

```
                 ( defcon1    )
                 | defld1     |
[label] TSTMSKI  { (defrf1)   } ,immdata2
                 | (reg1)     |
                 ( seg1,disp1 )
```

operand 1
Is a field to be tested. The length of the field must be either 1 or 2 bytes.

operand 2
Is the 1 or 2 bytes of immediate data to be used as the mask. The length of the mask determines the length of the operation.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	MZ	All tested bits are 0's or the mask bits are all 0's.
02	MX	The tested bits are mixed 1's and 0's.
04	ME	The tested field and the mask are identical.
08	MO	All tested bits are 1's.

Note: Condition code hex 04 is set with hex 08; it is never set alone. Consider this when writing an instruction that uses this code as the condition for a branch.

Program Checks (hex): 01, 02, 03, or 27 can be set.

Programming Notes: For example, the application program must now transmit the output field to the central processor or record the field on the diskette for later transmission. The decision is made by testing the link status bit in Segment 15. The example below shows the test. (Assume that a COPY DEFGMS is included in the application program to provide the field definition for GMSIND.)

```
TSTLNK     TSTMSKI     GMSIND,GMSILDM                    1
                       BRAN              MO,DISKWRT      2
                       BRANL             CPUWRT          3
```

1 Tests bit 0 of the Segment 15 indicator byte to determine the status of the link (see the COPY DEFGMS instruction in Appendix B for definitions of GMSIND and GMSILDM).

2 Branches to the diskette write routine if the tested bit is on (the link is down).

3 Branches to the central processor write routine if the tested bit is off (the link is up).

UPKFLD--Unpack Field

UPKFLD translates a fixed field of 4-bit binary codes into hexadecimal EBCDIC. The unpacked field can consist of the numbers 0 through 9 and letters A through F. The length of the unpacked field is twice the length of the field to be unpacked.

Refer to the PAKFLD instruction for the table of packed and unpacked equivalents.

Name	Operation	Operand

```
                              ( defcon2            )
                              | defld2             |
[label] UPKFLD    seg1,      <  (defrf2)            >
                              | (reg2)             |
                              ( seg2,disp2,len2    )
```

operand 1
Is a field in the specified segment that is to contain the unpacked field. The field must not be in Segment 14. The location of the field within the segment to contain the unpacked field is indicated by the primary field pointer (PFP). The field length indicator is ignored unless the operand 2 length is specified as 0. Following the execution of the instruction, the PFP points 1 byte past the end of the field, and the field length indicator is unchanged.

operand 2
Is a field to be unpacked. The length of the field to be unpacked is from 0 to 15, unless a form of register addressing is used, which allows a length ranging from 0 to 255. If 0 is specified, the field length indicator of operand 1 indicates the length of the operation.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, 03, or 27 can be set.

UPKSEG--Unpack Segment

UPKSEG translates a variable field of 4-bit binary codes into hexadecimal EBCDIC. The unpacked field can consist of the numbers 0 through 9 and letters A through F. The unpacked field is twice the length of the field to be unpacked. If the length of the field to be unpacked is equal to 0, no operation occurs.

Refer to the PAKFLD instruction for the table of packed and unpacked equivalents.

Name	Operation	Operand
[label]	UPKSEG	seg1,seg2

operand 1
Is a field in the specified segment to contain the unpacked data. The field must not be in Segment 14. If the operands refer to different segments, the location of the field to contain the unpacked field is indicated by the primary field pointer (PFP). If both operands refer to the same segment, the location of this field is indicated by the secondary field pointer (SFP); when the operation is completed, the primary field pointer points to the first byte past the end of the field. In either case, the field length indicator is ignored.

operand 2
Is a field in the specified segment containing the data to be unpacked. The PFP indicates the start of the field, and the field length indicator gives the length of the field to be unpacked. This length cannot exceed 255. The PFP is not changed when the instruction is executed.

Condition Codes: The code is not changed.

Program Checks (hex): 01, 02, or 03 may be set.

USEBASE--Use a Base Register for a DSECT

This instruction specifies that a register will be used as the base register for a DSECT.

Name	Operation	Operand
	USEBASE	label, { reg|* }

label

specifies the label of a LDSECT instruction.

reg

specifies a register that will be used as the base register when references are made to the field within the specified DSECT. This value overrides any previous specification made either on the LDSECT instruction or a previous USEBASE instruction.

An '*'

indicates that the register should be reset to the specification made on the LDSECT instruction.

VERIFY--Verify

VERIFY checks a field for either (1) a maximum and minimum field length, or (2) a maximum and minimum field length and EBCDIC numeric characters.

Name	Operation	Operand
[label]	VERIFY	seg1, { defcon2 | defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 1

Is a field in the specified segment to be verified. The start of the field is indicated by the primary field pointer, and the length by the field length indicator.

operand 2

Is a field containing the parameter list. The length associated with this operand is ignored, and the first 3 bytes are assumed to be the parameter list.

The format of the parameter list is shown below.

Type	Max	Min
0	1	2

Type

Is a 1-byte hexadecimal value that indicates the type of check that is to be made. If hex 01, a check is made to verify that the length of the field pointed to by the first operand is within the limits of the minimun and maximum. If hex 02, the check is the same as a type hex 01 check, but the field is also checked to verify that all characters are valid EBCDIC numerics. The first position only may be a minus sign; if it is a minus sign, it is not included in the check for length.

Note: Any other value specified for type defaults to hex 01.

Max

Is the maximum length of the field (0-255), excluding the minus sign for negative numeric fields.

Min

Is the minimum length of the field (0-255), excluding the minus sign for negative numeric fields.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	The field checked corresponds to the characteristics described by the parameter list.
02	IL	The field length is not within the limits described by the parameter list.
04	NN	The field is not numeric.

Program Checks (hex): 01, 02, or 27 can be set.

Programming Notes: In this example, VERIFY tests a field for both length and EBCDIC numeric characters. The result of the check is indicated by one or more condition codes. The application program tests the condition code to determine whether the field is valid or invalid. The instructions below show how the account number entered by the teller is tested for 7 numeric digits. Step 2 of the example sets the PFP and FLI of INPUTSEG as follows:

```
         SDEP          3382206 128.50 400. EOM*

                       PFP  FLI

ACCTCHK  DEFCON        X'020707'               1
SAVRTN   SETFPL        INPUTSEG,+F             2
         VERIFY        INPUTSEG,ACCTCHK        3
         BRAN          X'06',ERROR             4
```

1. Provides the testing criteria used by VERIFY. It specifies a check for length (a maximum and minimum of 7 characters) and all numerics.
2. Sets the PFP and FLI so that they define the field containing the account number.
3. Checks whether the account number consists of seven numeric digits. A condition code is set to indicate the result.
4. Changes the program flow if the account number was not within the limits described by the parameter list.

VIEW--VIEW APCALL/APRETURN Stack

The VIEW instruction returns information associated with an APCALL/APRETURN stack entry for a given station. Stack entry IDs correspond to segment space IDs.

The operand of VIEW locates a parameter list defined by the COPY DEFVUE instruction. Prior to VIEW execution the application program must initialize:

1. VUEREQ to X'01'

2. VUESTA to the number of the station for which information is to be returned

3. VUESTK to the stack ID for which information is to be returned. A value of X'00' indicates a request for the current stack entry of the specified station.

The VIEW instruction will set:

1. VUESTK to contain the requested stack ID (this is significant only if X'00' was used on input to request the 'current' entry)

2. VUEFG1 to contain flag bits as follows:

   a. VUEFL0M if set indicates that the entry is in use. If reset, then the entry is not in use and other fields returned by VIEW are residual (that is, values left from the last use of the stack entry).

   b. VUEFL1M if set indicates the entry is permanent.

   c. VUEFL2M if set indicates that the 'B' set of registers (Segment 0) is active. If reset, it indicates that the 'A' set of registers is active.

3. VUEUIC to the user instruction counter.

4. VUELSB to the bottom of the SMS link stack.

5. VUELSE to the top of the SMS link stack.

6. VUEDEL to the saved value of SMSDEL (delimiter table).

7. VUEPNT to the parent segment stack ID of the requested stack entry. If the returned stack ID is X'01', then the parent segment space ID is set to X'00' indicating no parent exists.

8. VUEFG2 to contain flag bits as follows:

    a. VUERTF if set indicates the application program is transient.

    b. VUEFAC if set indicates that the application program can be called by an APCALL instruction.

9. VUEPID to the application program name.

An application program could, for example, fetch the calling program name and instruction counter by executing a VIEW instruction with X'00' to get the current stack's parent ID, then execute a VIEW requesting this parent ID.

The format of the VIEW instruction is:

Name	Operation	Operand
[label]	VIEW	{ defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 2

Is the field containing the parameter list. The field must not be in Segment 14. The length associated with the field is ignored, as the parameter list is assumed to be 26 bytes long.

Condition Codes: One of the following is set:

Hex Code	Possible Mnemonic	Explanation
01	OK	Normal completion (this includes a stack entry which is not in use)
02	ID	Invalid station number.
04		Invalid segment space ID.

Program Checks (hex): 01, 02, or 11 can be set.

Appendix A. Machine Instruction Formats

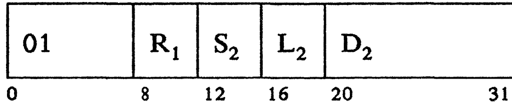
This appendix contains the machine instruction formats of the 4700 assembler language instructions that are described in this book. The machine instructions are listed in operation code sequence within an alphabetic sequence by instruction mnemonic. In other words, the ADDFLD instruction is first and the 01 operation code of the ADDFLD instruction appears before the 73 operation code.

The following symbols appear in this appendix:

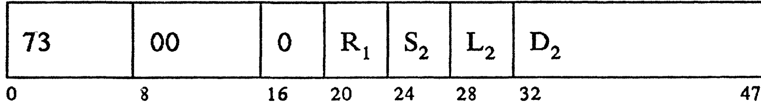
Symbol	Meaning
A(—)	Address of "—" (in adcon form)
$\mathbf{D_n}$	Displacement
$\mathbf{I_n}$	Immediate data
L	Length
M	Mask
N	Bit number
P	Field pointer
$\mathbf{R_n}$	Register
$\mathbf{S_n}$	Segment
T	First bit in field is a wait modifier (0 for wait, 1 for no wait); remaining bits are logical device address (LDA)
W	First bit in field is a wait modifier (0 for wait, 1 for no wait); remaining bits are reserved
X	Code - see Volume 1
Y	File - Field is set as follows (see Volume 2 for explanation):

Hex Digit	*Mnemonic*
0	C
1	TF1
2	TF2
3	TF3
4	TF4
5	PLR
6	PBN
9	DSID
D	A
E	P
F	L

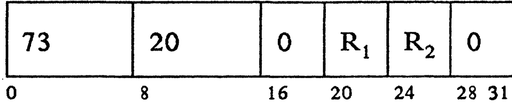
ADDFLD



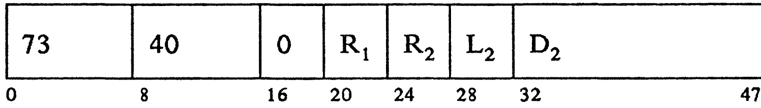
ADDFLD



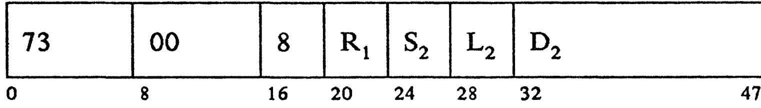
ADDFLD



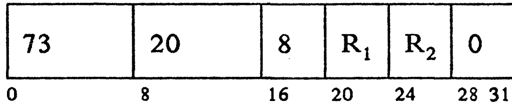
ADDFLD



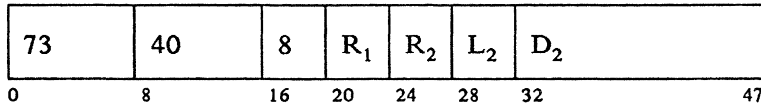
ADDFLDL



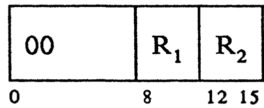
ADDFLDL



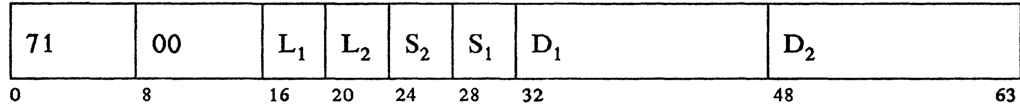
ADDFLDL



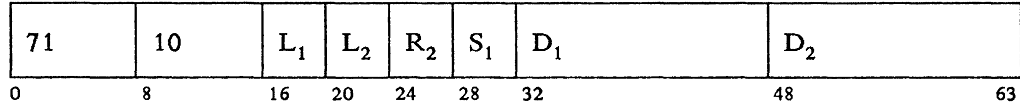
ADDREG



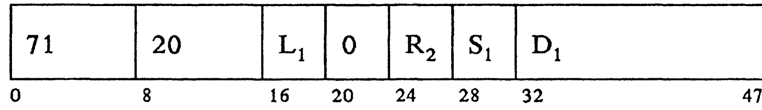
ADDZ



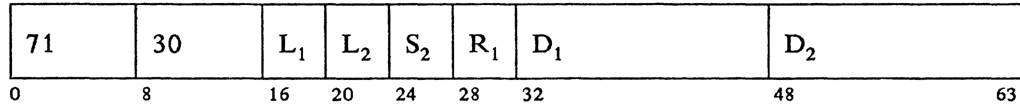
ADDZ



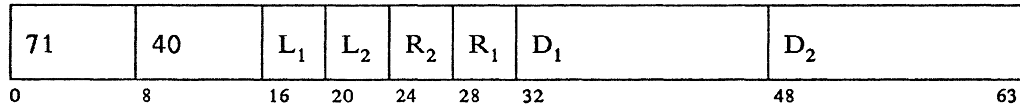
ADDZ



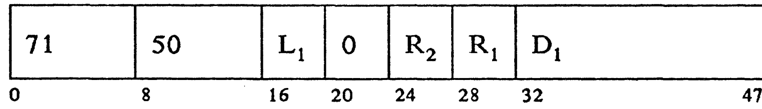
ADDZ



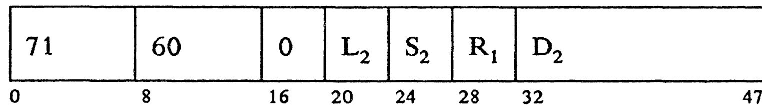
ADDZ



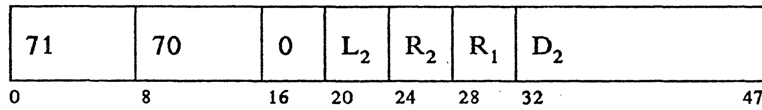
ADDZ



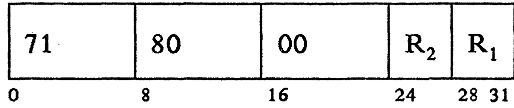
ADDZ



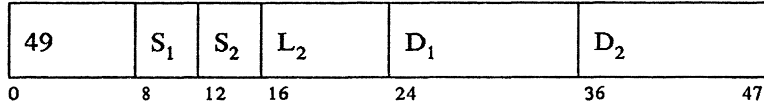
ADDZ



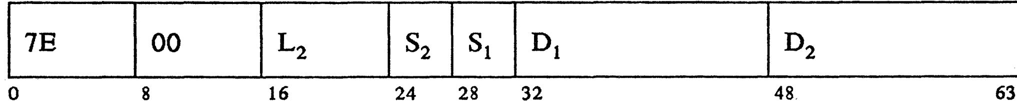
ADDZ



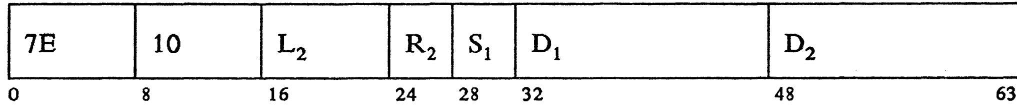
AND



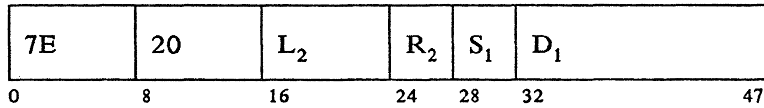
AND



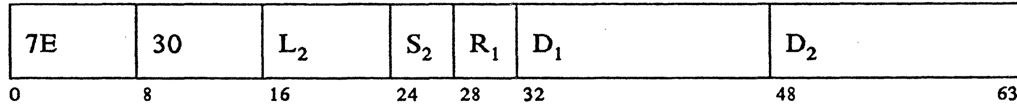
AND



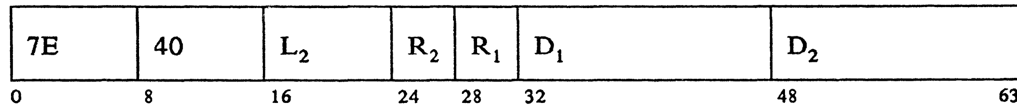
AND



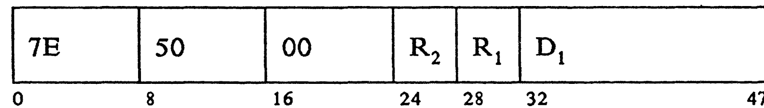
AND



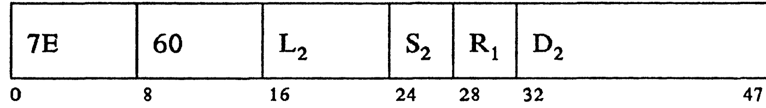
AND



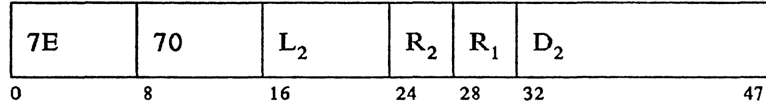
AND



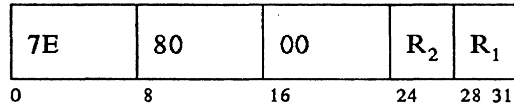
AND



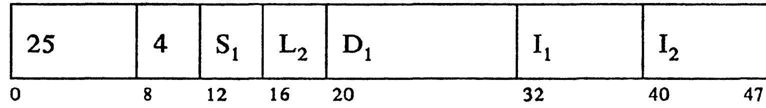
AND



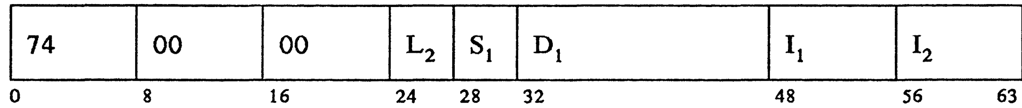
AND



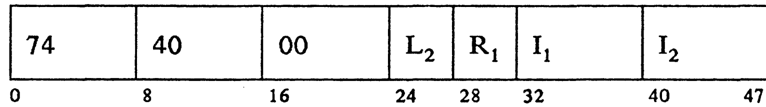
ANDI



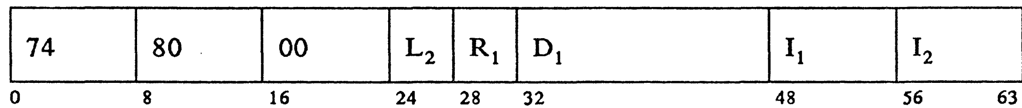
ANDI



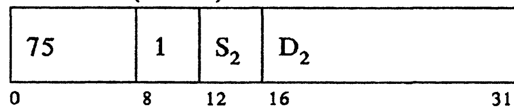
ANDI



ANDI



APCALL (WAIT)



APCALL (WAIT)

7A	0B	00	R_2	0
0	8	16	24	28 31

APCALL (NOWAIT)

7A	0C	00	R_2	0
0	8	16	24	28 31

APCALL (WAIT)

7A	1B	00	R_2	0	D_2
0	8	16	24	28	32 47

APCALL(NOWAIT)

7A	1C	00	R_2	0	D_2
0	8	16	24	28	32 47

APRETURN

75	20
0	8 15

BRAN

A	M	00	Address
0	4	8	16 31

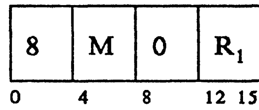
BRANL

B	M	R_2	0	Address
0	4	8	12	16 31

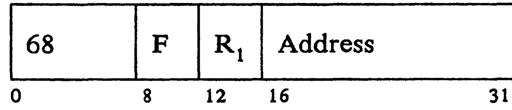
BRANLR

9	M	R_1	R_2
0	4	8	12 15

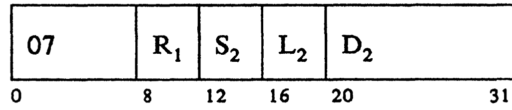
BRANR



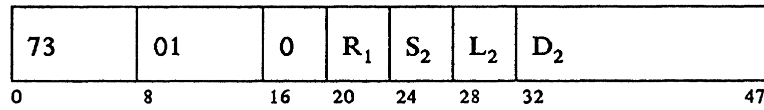
BRANX



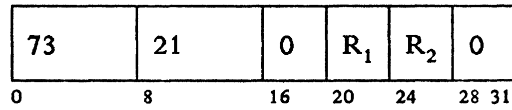
CAFLD



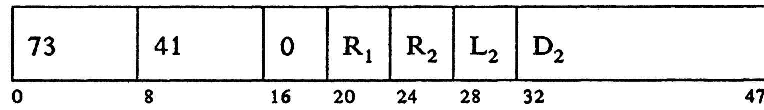
CAFLD



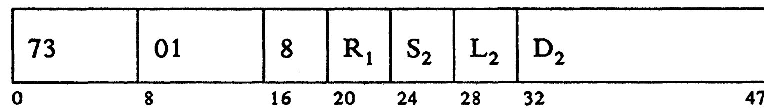
CAFLD



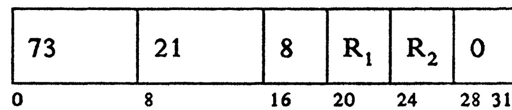
CAFLD



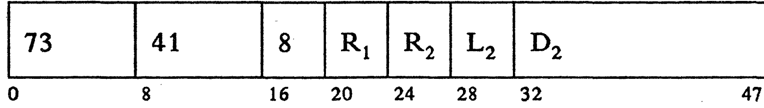
CAFLDL



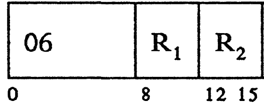
CAFLDL



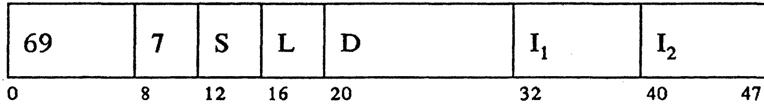
CAFLDL



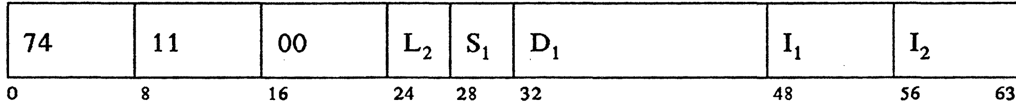
CAREG



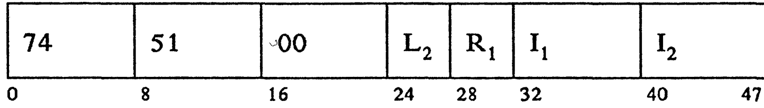
CCDI



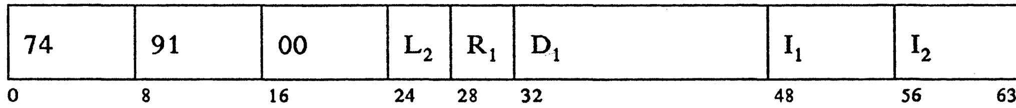
CCDI



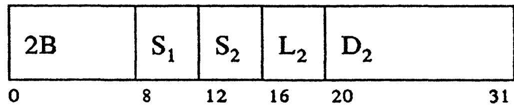
CCDI



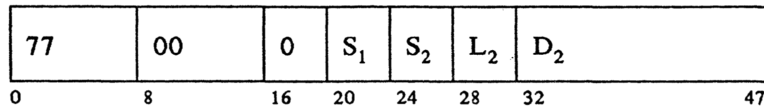
CCDI



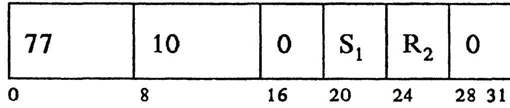
CCFLD



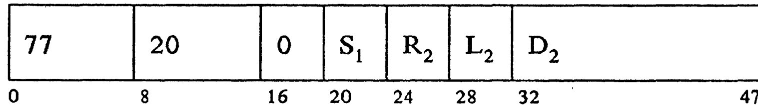
CCFLD



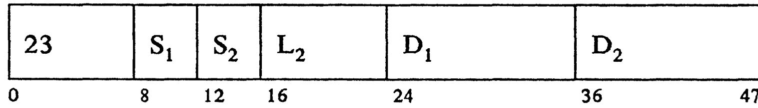
CCFLD



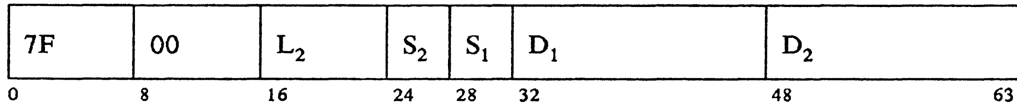
CCFLD



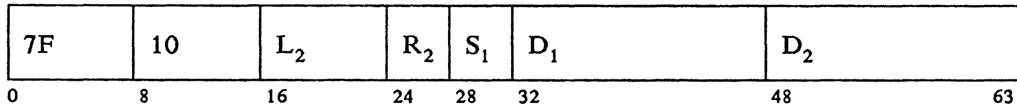
CCFXD



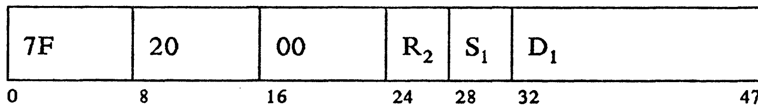
CCFXD



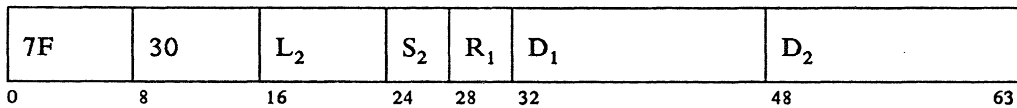
CCFXD



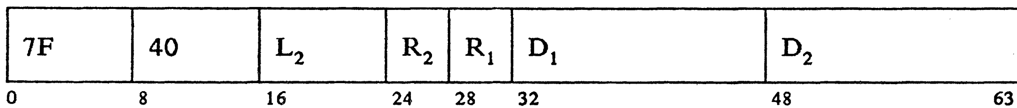
CCFXD



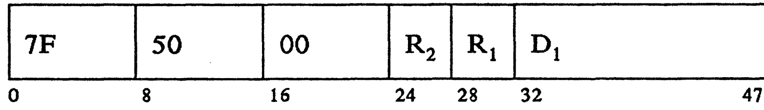
CCFXD



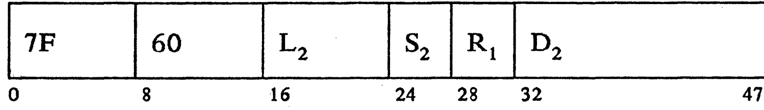
CCFXD



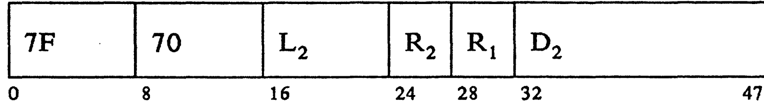
CCFXD



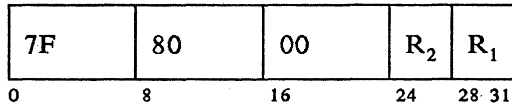
CCFXD



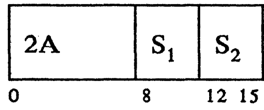
CCFXD



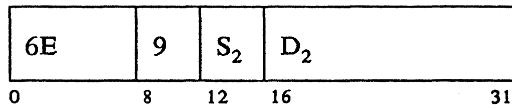
CCFXD



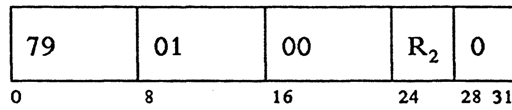
CCSEG



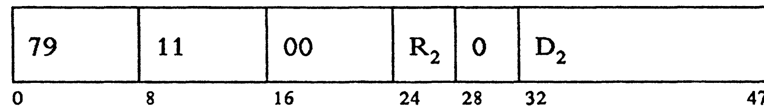
COMP



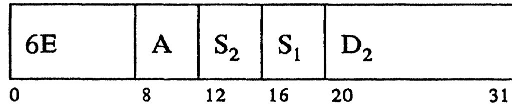
COMP



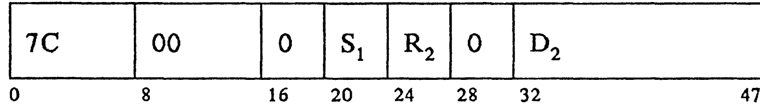
COMP



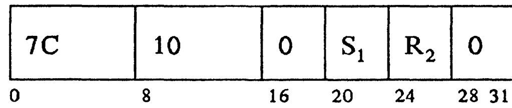
COMPTB



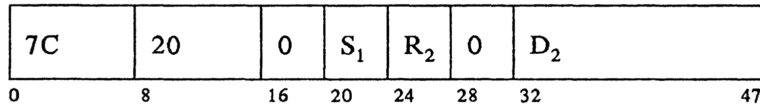
COMPTB



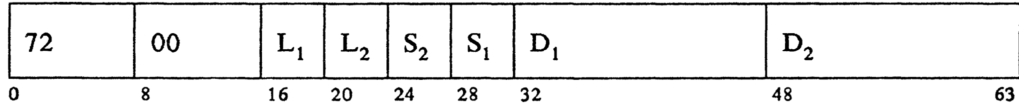
COMPTB



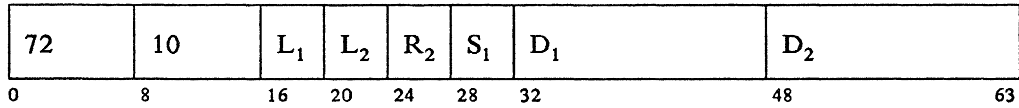
COMPTB



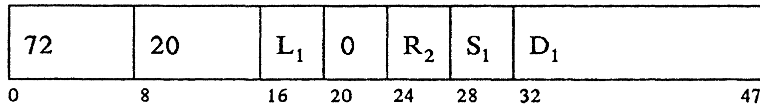
COMPZ



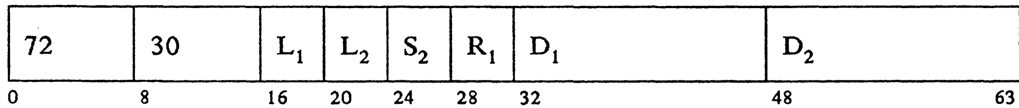
COMPZ



COMPZ



COMPZ



COMPZ

72	40	L_1	L_2	R_2	R_1	D_1	D_2
0	8	16	20	24	28	32	48 63

COMPZ

72	50	L_1	0	R_2	R_1	D_1
0	8	16	20	24	28	32 47

COMPZ

72	60	0	L_2	S_2	R_1	D_2
0	8	16	20	24	28	32 47

COMPZ

72	70	0	L_2	R_2	R_1	D_2
0	8	16	20	24	28	32 47

COMPZ

72	80	00	R_2	R_1
0	8	16	24	28 31

CRETN

75	3	S_2	D_2
0	8	12	16 31

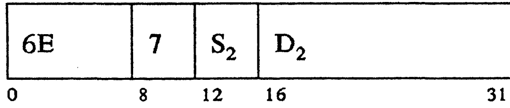
CRETN

79	2A	00	R_2	0
0	8	16	24	28 31

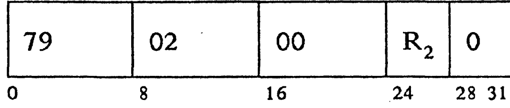
CRETN

79	3A	00	R_2	0	D_2
0	8	16	24	28	32 47

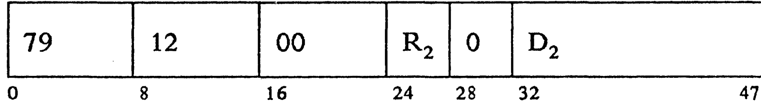
DECOMP



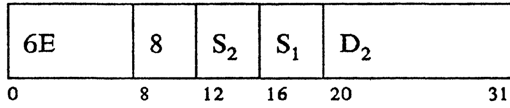
DECOMP



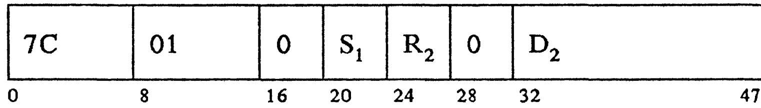
DECOMP



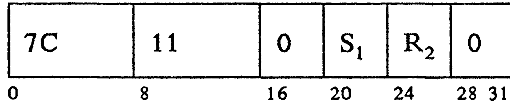
DECOMPTB



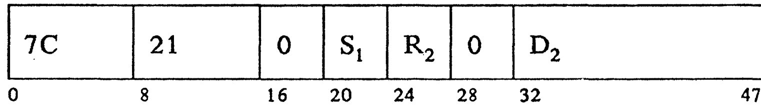
DECOMPTB



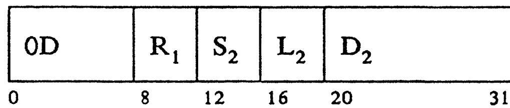
DECOMPTB



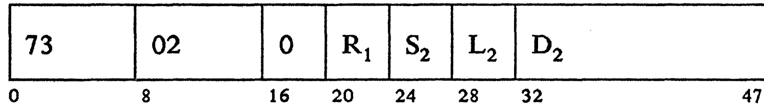
DECOMPTB



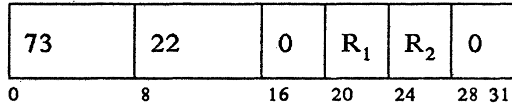
DIVFLD



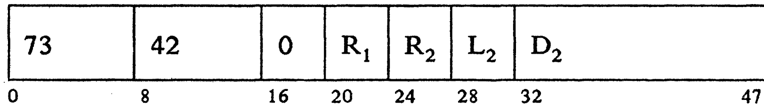
DIVFLD



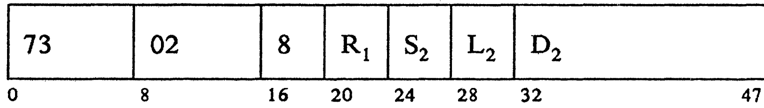
DIVFLD



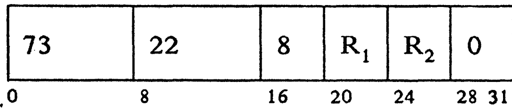
DIVFLD



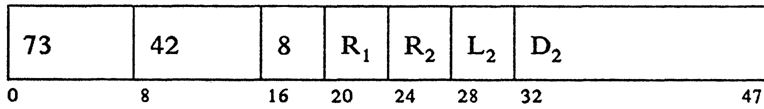
DIVFLDL



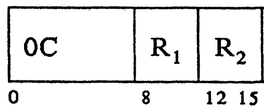
DIVFLDL



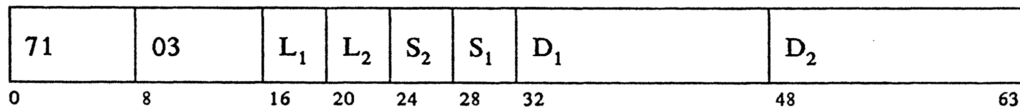
DIVFLDL



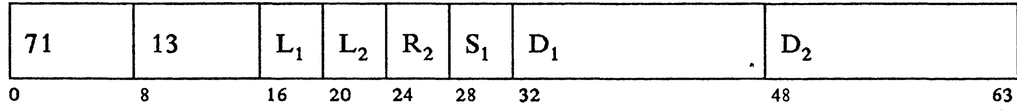
DIVREG



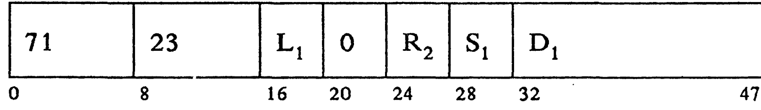
DIVZ



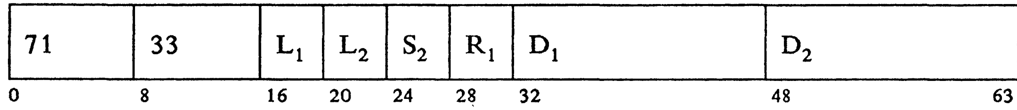
DIVZ



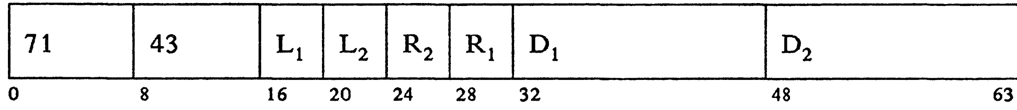
DIVZ



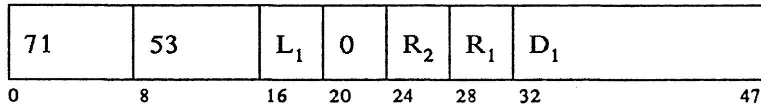
DIVZ



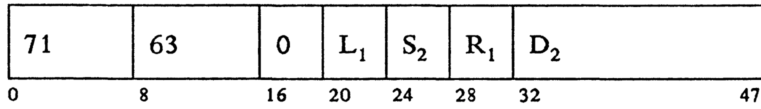
DIVZ



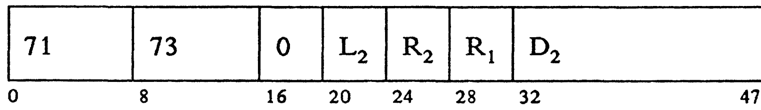
DIVZ



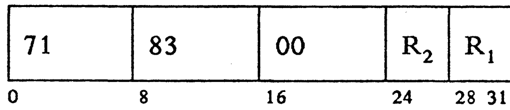
DIVZ



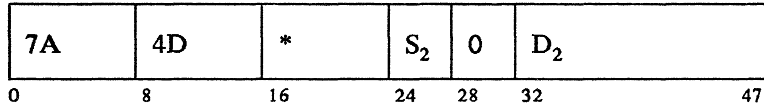
DIVZ



DIVZ

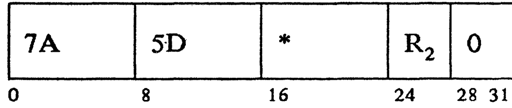


DTACCESS



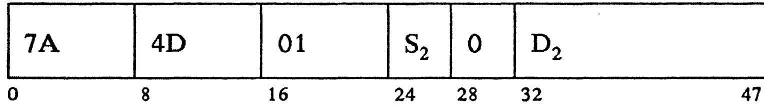
* = 00 if the WAIT Parameter = Y
08 if the WAIT Parameter = N

DTACCESS

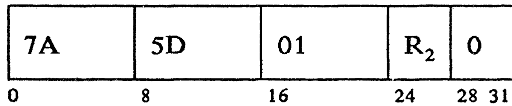


* = 00 if the WAIT Parameter = Y
08 if the WAIT Parameter = N

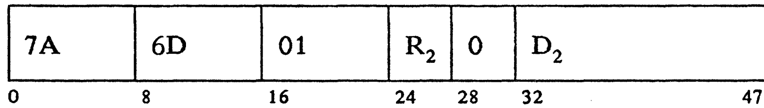
DTAFREE



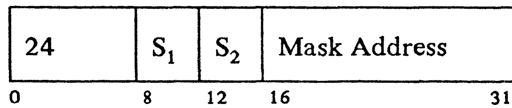
DTAFREE



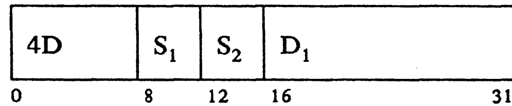
DTAFREE



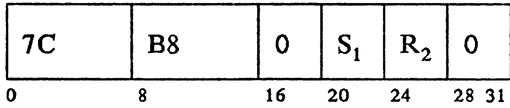
EDIT



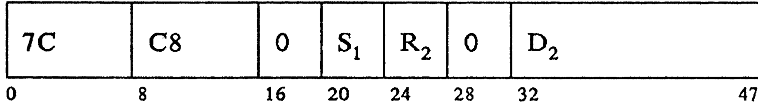
ERRLOG



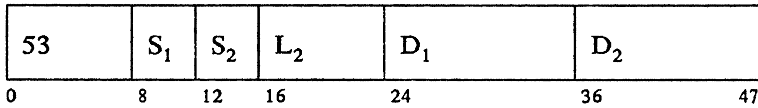
ERRLOG



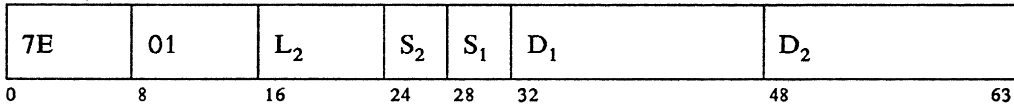
ERRLOG



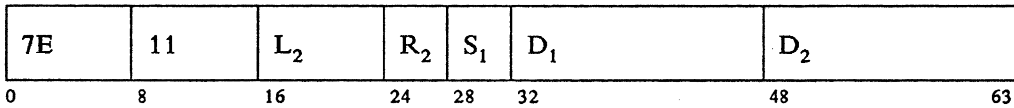
EXOR



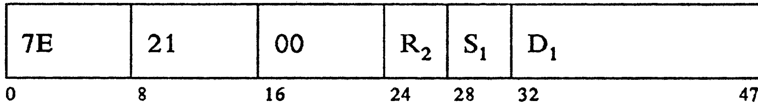
EXOR



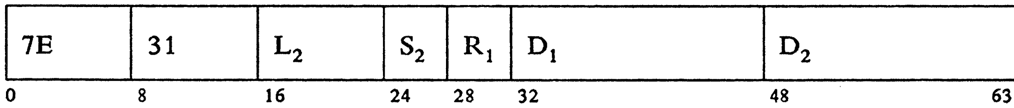
EXOR



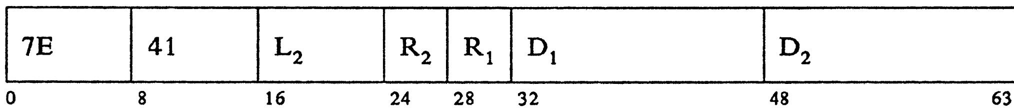
EXOR



EXOR



EXOR



EXOR

7E	51	00	R_2	R_1	D_1
0	8	16	24	28	32 — 47

EXOR

7E	61	L_2	S_2	R_1	D_2
0	8	16	24	28	32 — 47

EXOR

7E	71	L_2	R_2	R_1	D_2
0	8	16	24	28	32 — 47

EXOR

7E	81	00	R_2	R_1
0	8	16	24	28 31

EXORI

25	1	S_1	L_2	D_1	I_1	I_2
0	8	12	16	20	32	40 — 47

EXORI

74	02	00	L_2	S_1	D_1	I_1	I_2
0	8	16	24	28	32	48	56 — 63

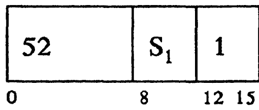
EXORI

74	42	00	L_2	R_1	I_1	I_2
0	8	16	24	28	32	40 — 47

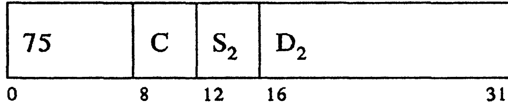
EXORI

74	82	00	L_2	R_1	D_1	I_1	I_2
0	8	16	24	28	32	48	56 — 63

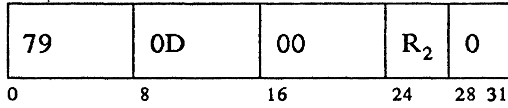
EXPS



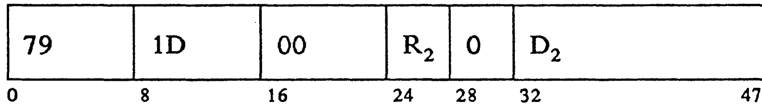
FINDAP



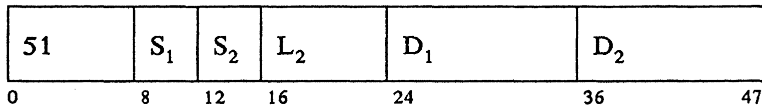
FINDAP



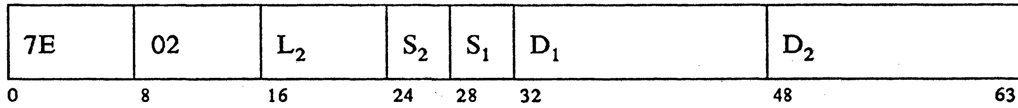
FINDAP



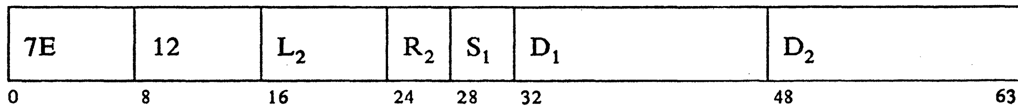
INOR



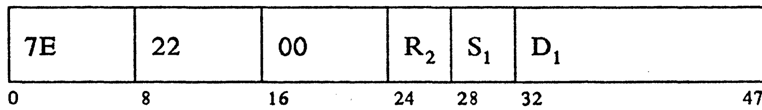
INOR



INOR



INOR



INOR

7E	32	L_2	S_2	R_1	D_1	D_2

0 8 16 24 28 32 48 63

INOR

7E	42	L_2	R_2	R_1	D_1	D_2

0 8 16 24 28 32 48 63

INOR

7E	52	00	R_2	R_1	D_1

0 8 16 24 28 32 47

INOR

7E	62	L_2	S_2	R_1	D_2

0 8 16 24 28 32 47

INOR

7E	72	L_2	R_2	R_1	D_2

0 8 16 24 28 32 47

INOR

7E	82	00	R_2	R_1

0 8 16 24 28 31

INORI

25	0	S_1	L_2	D_1	I_1	I_2

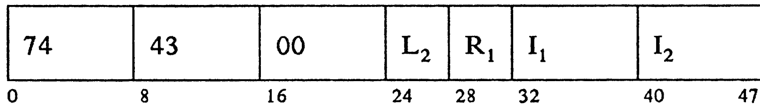
0 8 12 16 20 32 40 47

INORI

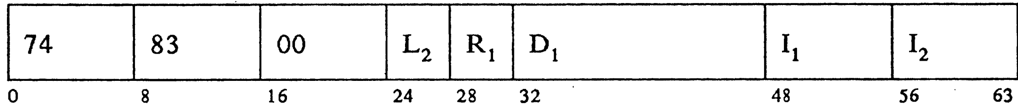
74	03	00	L_2	S_1	D_1	I_1	I_2

0 8 16 24 28 32 48 56 63

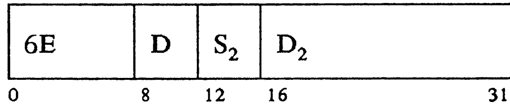
INORI



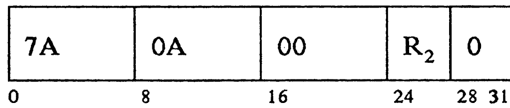
INORI



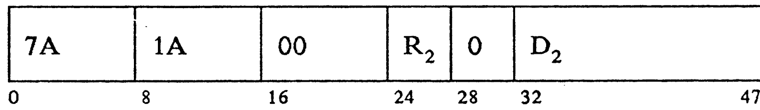
INTMR



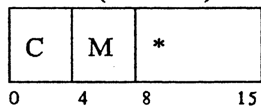
INTMR



INTMR

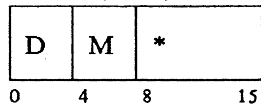


JUMP (forward)



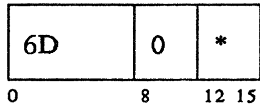
*=number of bytes being jumped

JUMP (back)



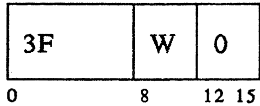
*=number of bytes being jumped

LCHAP

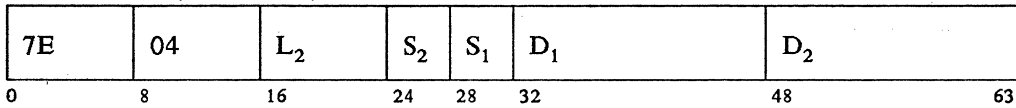


*Bits 12-13 = 0 for off, 2 for on, 1 for next; Bits 14-15 are reserved.

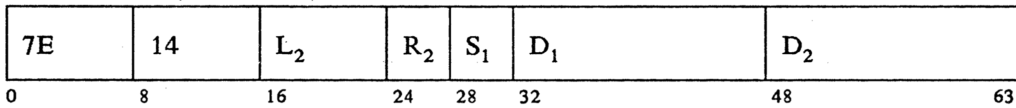
LCHECK (ST)



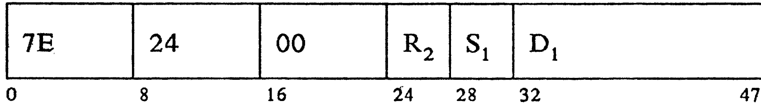
LCONVERT (TOBYTES)



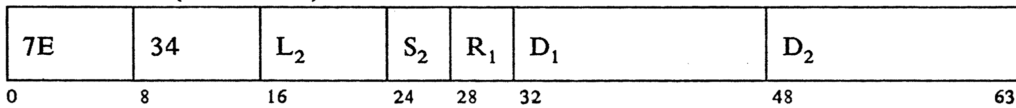
LCONVERT (TOBYTES)



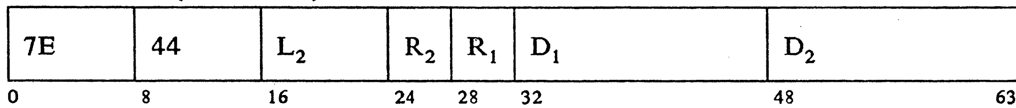
LCONVERT (TOBYTES)



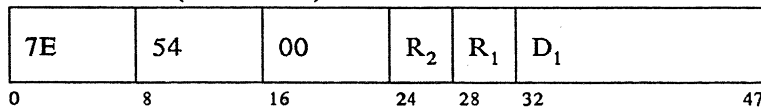
LCONVERT (TOBYTES)



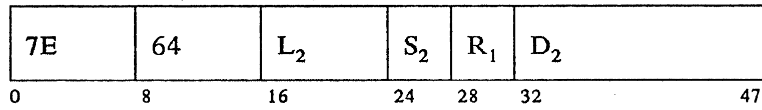
LCONVERT (TOBYTES)



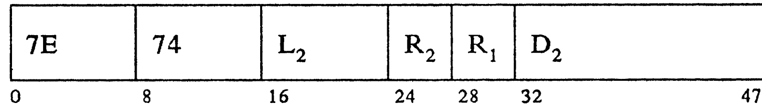
LCONVERT (TOBYTES)



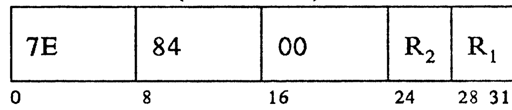
LCONVERT (TOBYTES)



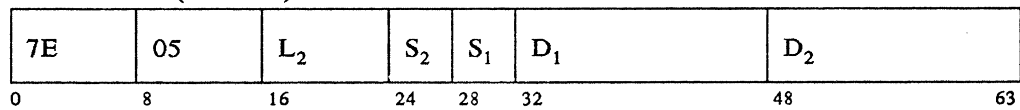
LCONVERT (TOBYTES)



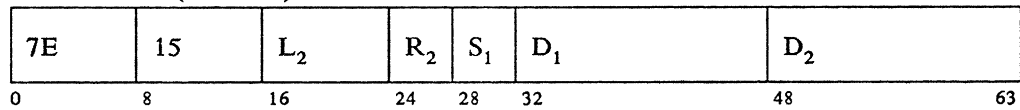
LCONVERT (TOBYTES)



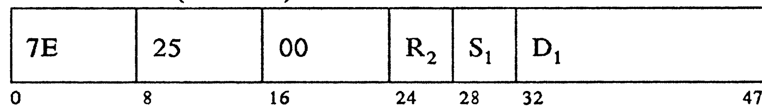
LCONVERT (TOBITS)



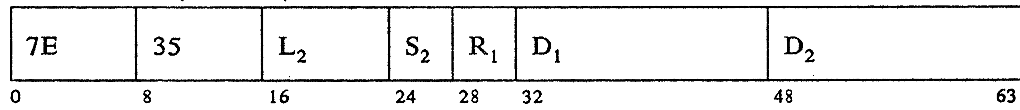
LCONVERT (TOBITS)



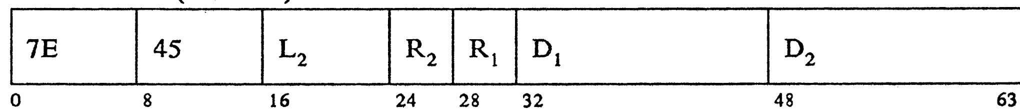
LCONVERT (TOBITS)



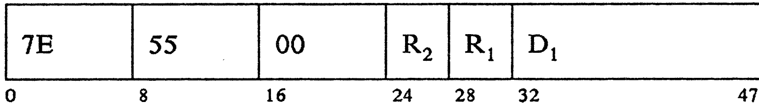
LCONVERT (TOBITS)



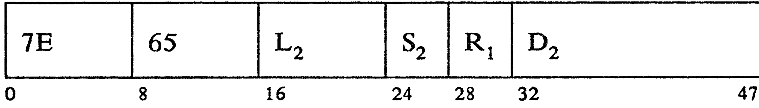
LCONVERT (TOBITS)



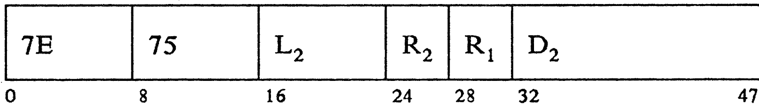
LCONVERT (TOBITS)



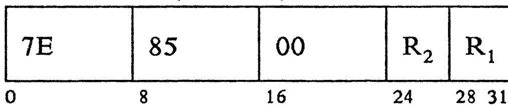
LCONVERT (TOBITS)



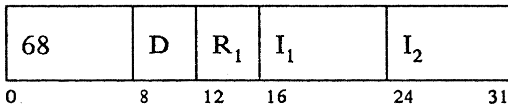
LCONVERT (TOBITS)



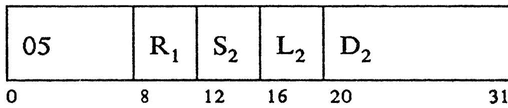
LCONVERT (TOBITS)



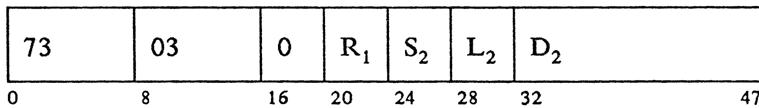
LDDI



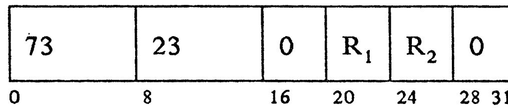
LDFLD



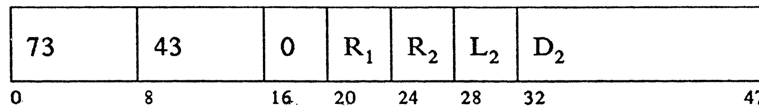
LDFLD



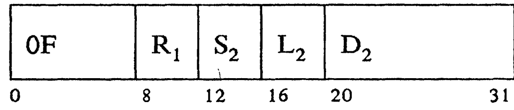
LDFLD



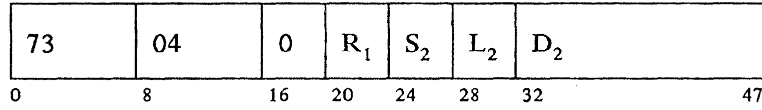
LDFLD



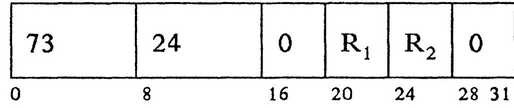
LDFLDC



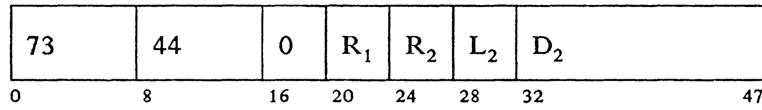
LDFLDC



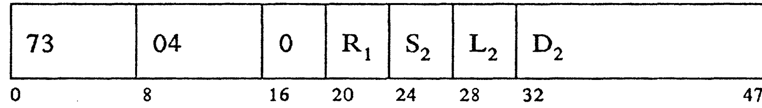
LDFLDC



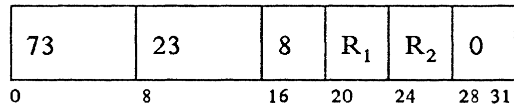
LDFLDC



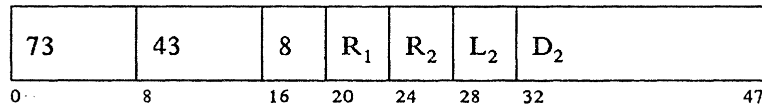
LDFLDC



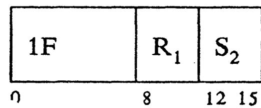
LDFLDL



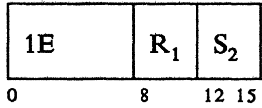
LDFLDL



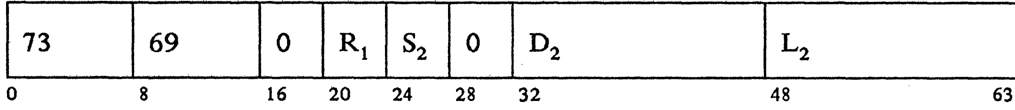
LDFP



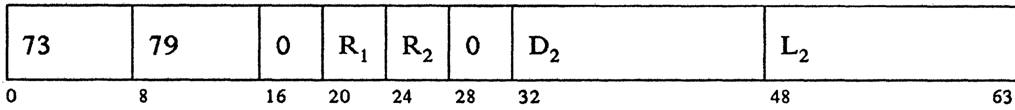
LDLN



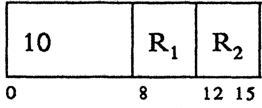
LDRA



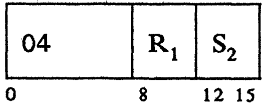
LDRA



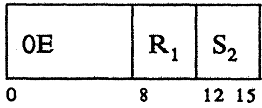
LDREG



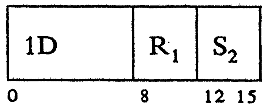
LDSEG



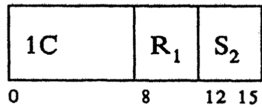
LDSEGC



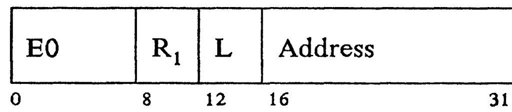
LDSEGLN



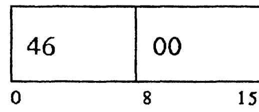
LDSFP



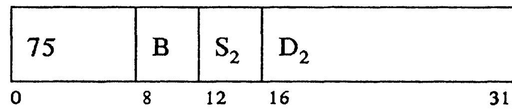
LEXEC



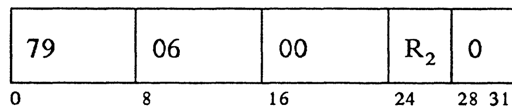
LEXIT



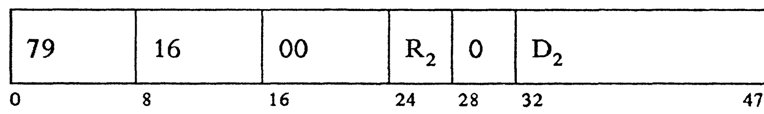
LHRT



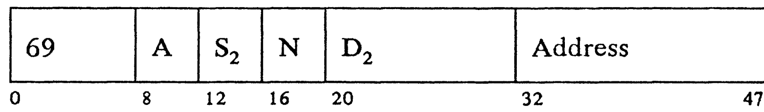
LHRT



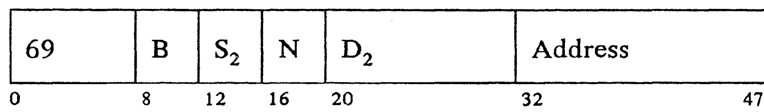
LHRT



LIFOFF

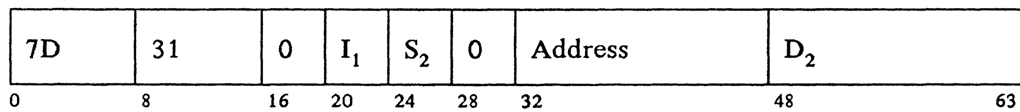


LIFOFF

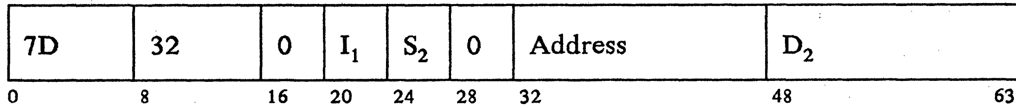


Change bit switch (N) if no branch taken.

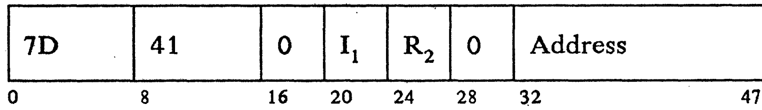
LIFOFF



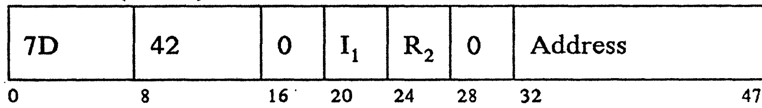
LIFOFF (ELSE)



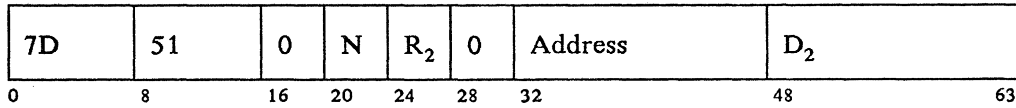
LIFOFF



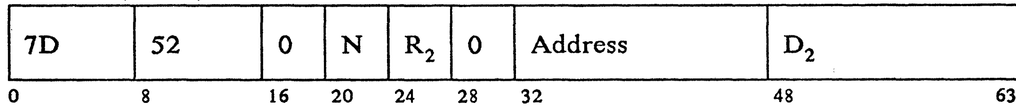
LIFOFF (ELSE)



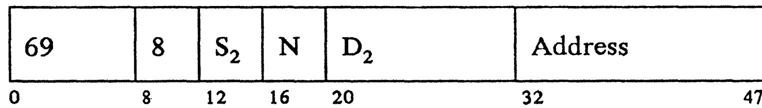
LIFOFF



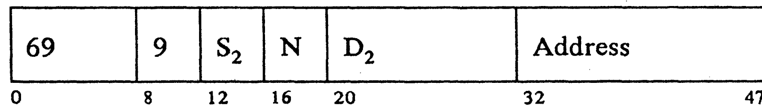
LIFOFF (ELSE)



LIFON

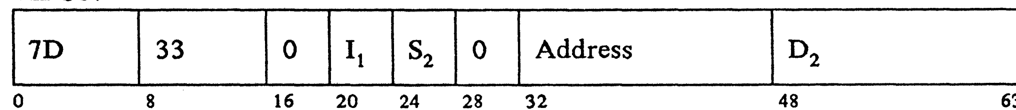


LIFON

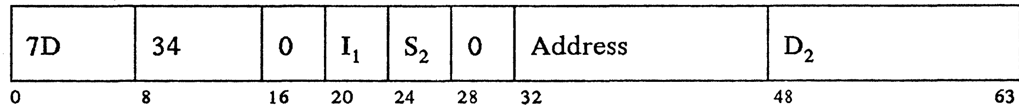


Change bit switch (N) if no branch taken.

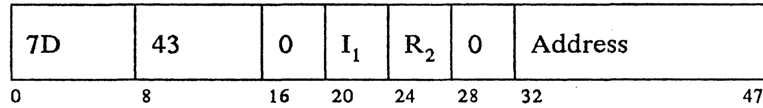
LIFON



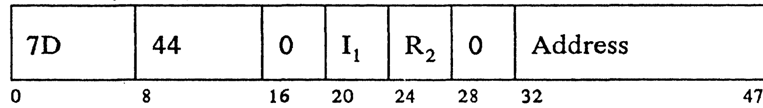
LIFON (ELSE)



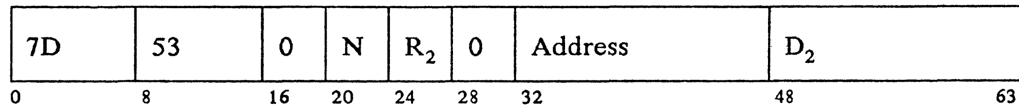
LIFON



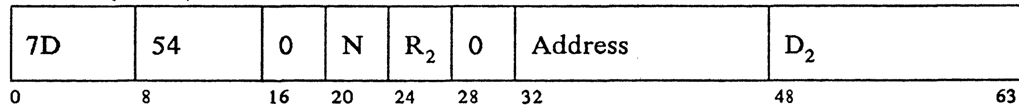
LIFON (ELSE)



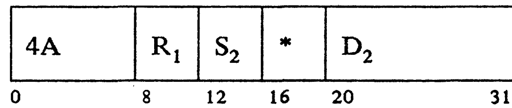
LIFON



LIFON (ELSE)

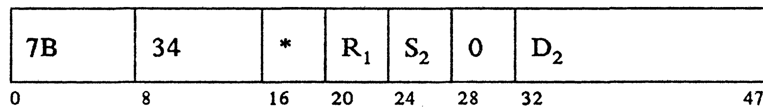


LLOAD



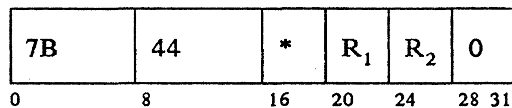
*Mask – 0 for normal LLOAD, 1 for expanded LLOAD

LLOAD



*0 for normal LLOAD; 1 for expanded LLOAD

LLOAD



*0 for normal LLOAD; 1 for expanded LLOAD

LLOAD

7B	54	*	R_1	R_2	0	D_2

0 8 16 20 24 28 32 47

*0 for normal LLOAD; 1 for expanded LLOAD

LMERGE

5C	1	S_2

0 8 12 15

LMERGE

5D	1	S_2	D_2

0 8 12 16 31

LMERGE

79	28	00	R_2	0

0 8 16 24 28 31

LMERGE

79	38	00	R_2	0	D_2

0 8 16 24 28 32 47

LPOST

26	80

0 8 15

LREAD (ST)

43	0	S_2

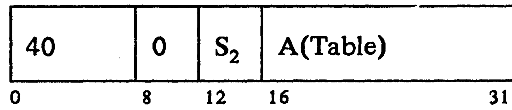
0 8 12 15

LRETURN

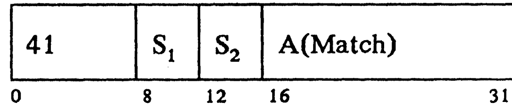
44	00

0 8 15

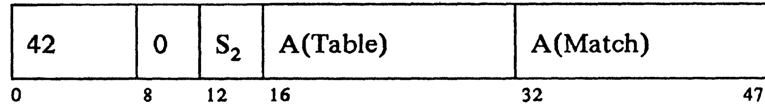
LSEEK



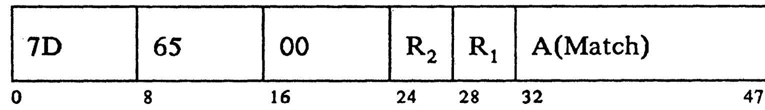
LSEEK



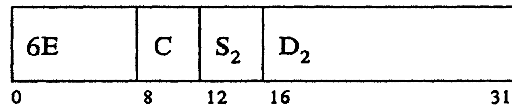
LSEEK



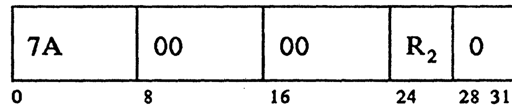
LSEEK



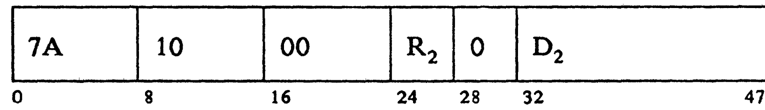
LSEEKP



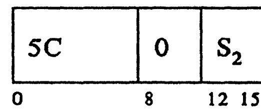
LSEEKP



LSEEKP



LSORT



LSORT

5D	0	S_2	D_2
0	8	12	16 … 31

LSORT

7A	21	00	R_2	0
0	8	16	24	28 31

LSORT

7A	31	00	R_2	0	D_2
0	8	16	24	28	32 … 47

LTIME (ADJ)

75	A	S_2	D_2
0	8	12	16 … 31

LTIME (ADJ)

7A	09	00	R_2	0
0	8	16	24	28 31

LTIME (ADJ)

7A	19	00	R_2	0	D_2
0	8	16	24	28	32 … 47

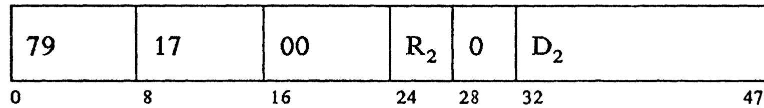
LTIME (GET)

75	6	S_2	D_2
0	8	12	16 … 31

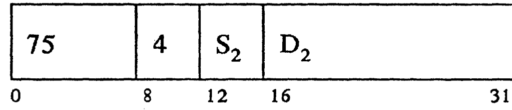
LTIME (GET)

79	07	00	R_2	0
0	8	16	24	28 31

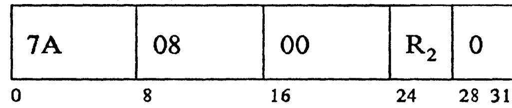
LTIME (GET)



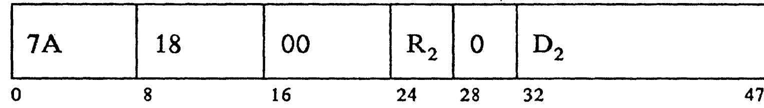
LTIME (SET)



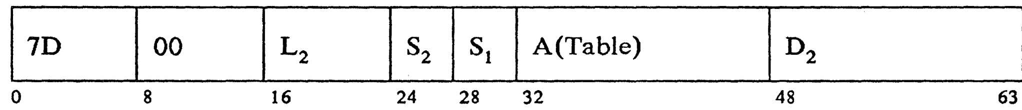
LTIME (SET)



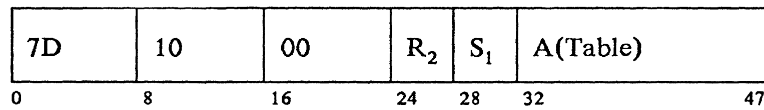
LTIME (SET)



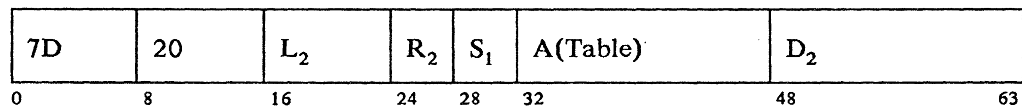
LTIMEV



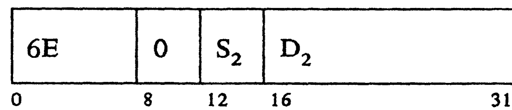
LTIMEV



LTIMEV



LTRT



LTRT

79	09	00	R_2	0	
0	8	16	24	28	31

LTRT

79	19	00	R_2	0	D_2	
0	8	16	24	28	32	47

LWAIT

26	40	
0	8	15

LWRITE (ST)

3C	0	S_2	
0	8	12	15

LWRITE (ST)

3E	0	S_2	L_2	D_2	00	
0	8	12	16	28	40	47

LWRITE (ST)

78	38	00	S_2	0	D_2	L_2	
0	8	16	24	28	32	48	63

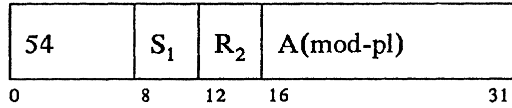
LWRITE (ST)

78	48	00	R_2	0	
0	8	16	24	28	31

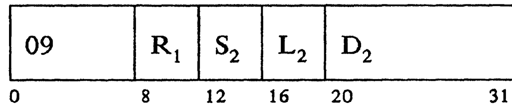
LWRITE (ST)

78	58	00	R_2	0	D_2	L_2	
0	8	16	24	28	32	48	63

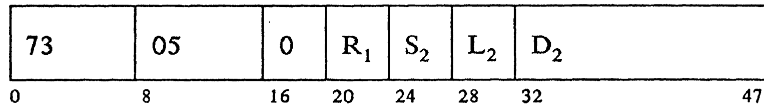
MODCHK



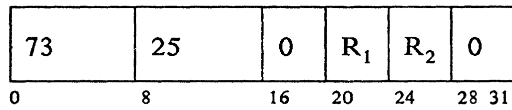
MPYFLD



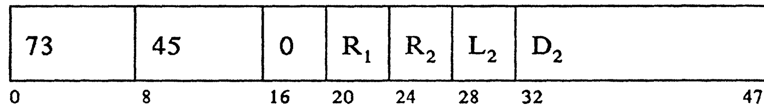
MPYFLD



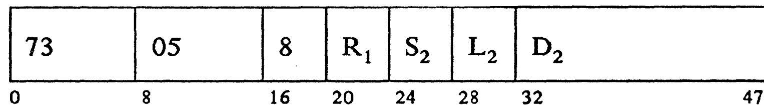
MPYFLD



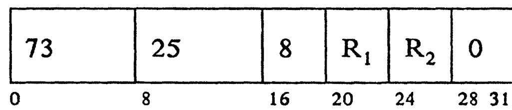
MPYFLD



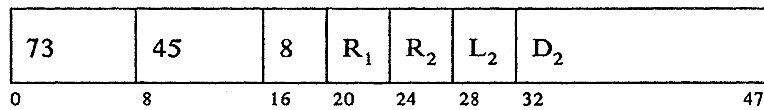
MPYFLDL



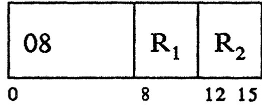
MPYFLDL



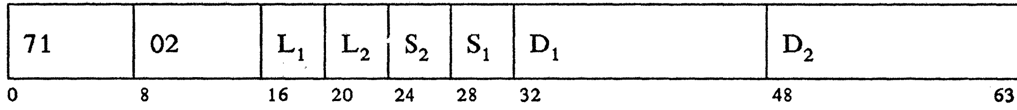
MPYFLDL



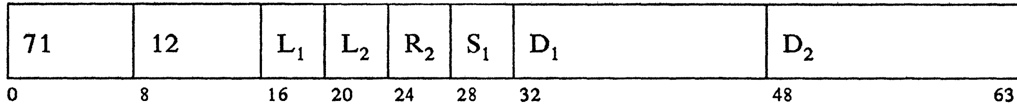
MPYREG



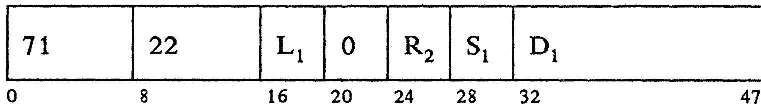
MPYZ



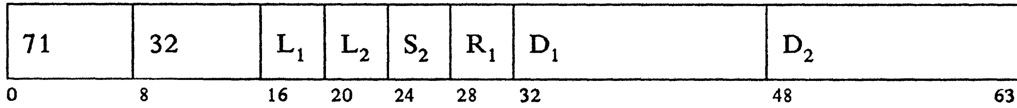
MPYZ



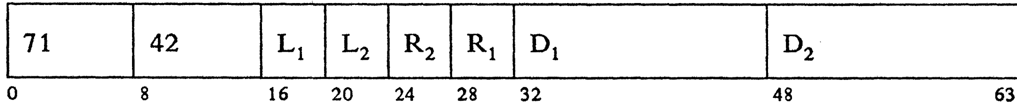
MPYZ



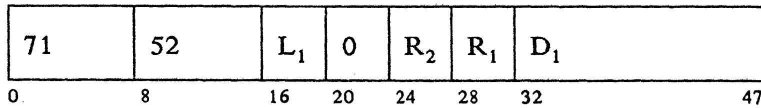
MPYZ



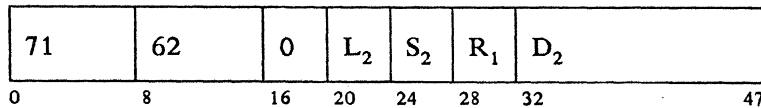
MPYZ



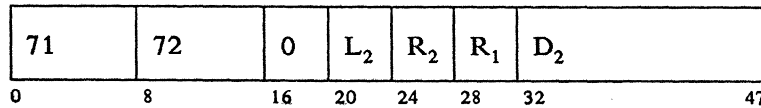
MPYZ



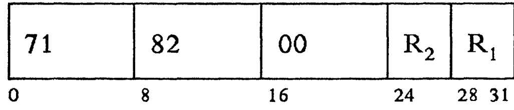
MPYZ



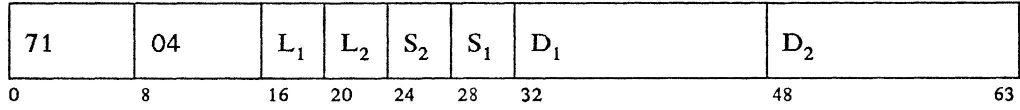
MPYZ



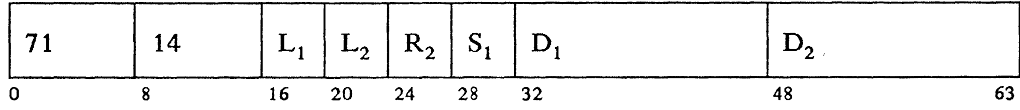
MPYZ



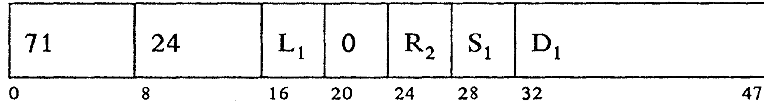
MVCZ



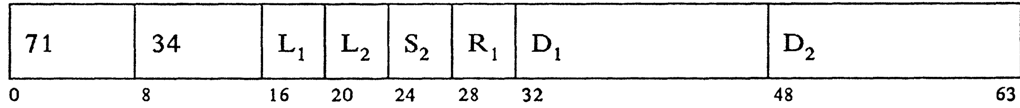
MVCZ



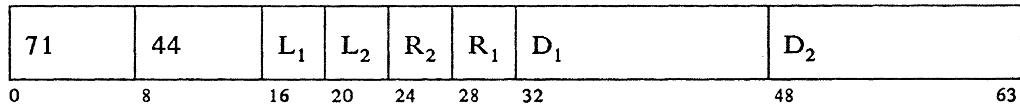
MVCZ



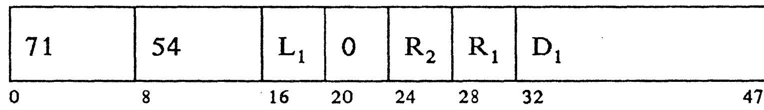
MVCZ



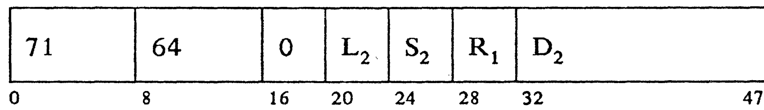
MVCZ



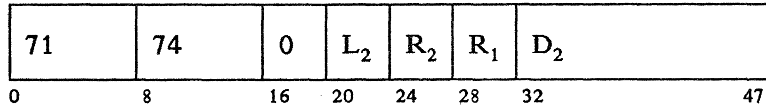
MVCZ



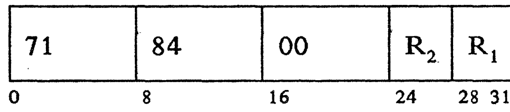
MVCZ



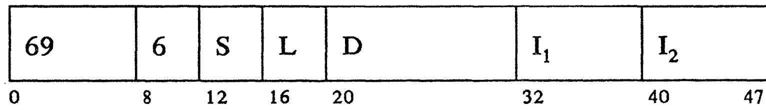
MVCZ



MVCZ

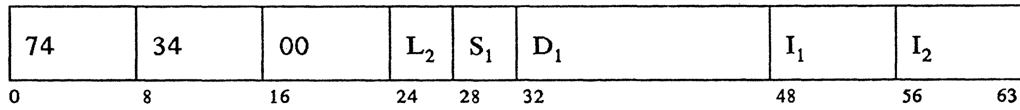


MVDI

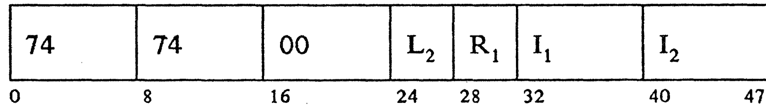


If only one byte of data is specified, X'00' is added in bits 40—47.

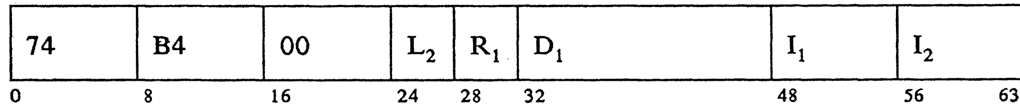
MVDI



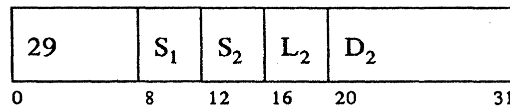
MVDI



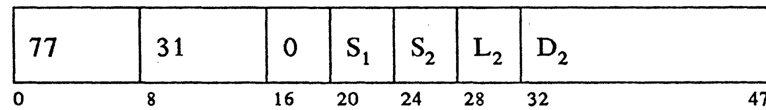
MVDI



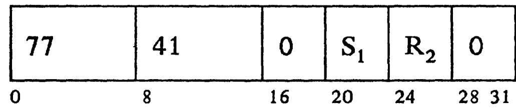
MVFLD



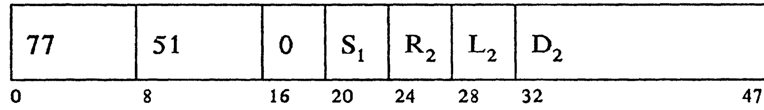
MVFLD



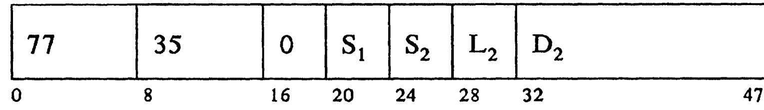
MVFLD



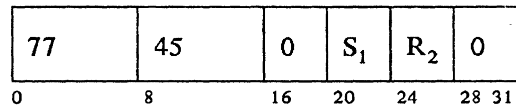
MVFLD



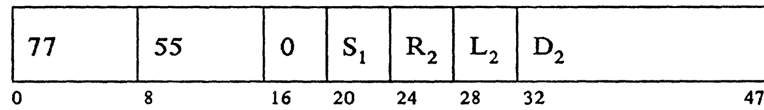
MVFLDR



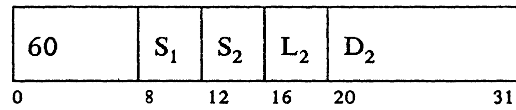
MVFLDR



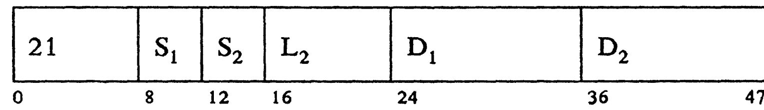
MVFLDR



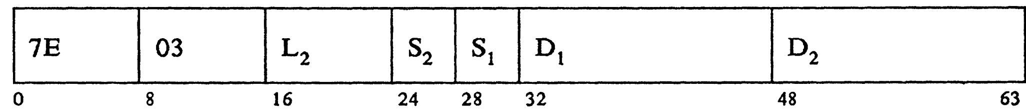
MVFLDR



MVFXD



MVFXD



MVFXD

7E	13	L_2	R_2	S_1	D_1	D_2
0	8	16	24	28	32	48 63

MVFXD

7E	23	00	R_2	S_1	D_1
0	8	16	24	28	32 47

MVFXD

7E	33	L_2	S_2	R_1	D_1	D_2
0	8	16	24	28	32	48 63

MVFXD

7E	43	L_2	R_2	R_1	D_1	D_2
0	8	16	24	28	32	48 63

MVFXD

7E	53	00	R_2	R_1	D_1
0	8	16	24	28	32 47

MVFXD

7E	63	L_2	S_2	R_1	D_2
0	8	16	24	28	32 47

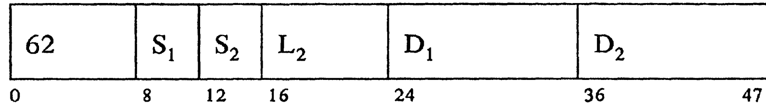
MVFXD

7E	73	L_2	R_2	R_1	D_2
0	8	16	24	28	32 47

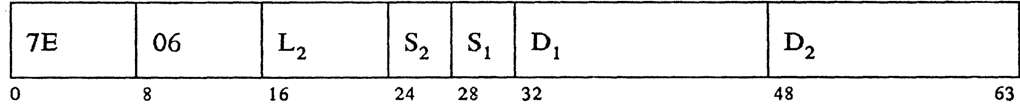
MVFXD

7E	83	00	R_2	R_1
0	8	16	24	28 31

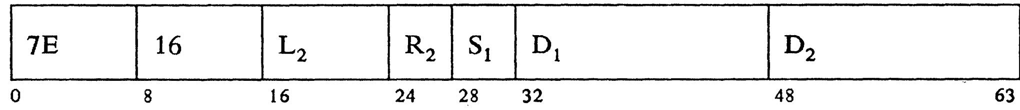
MVFXDR



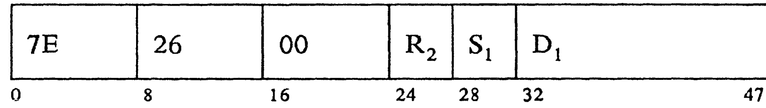
MVFXDR



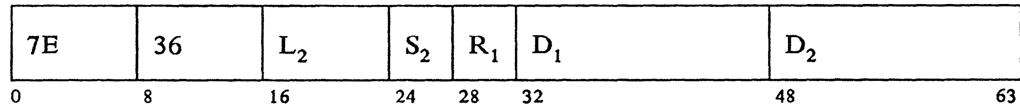
MVFXDR



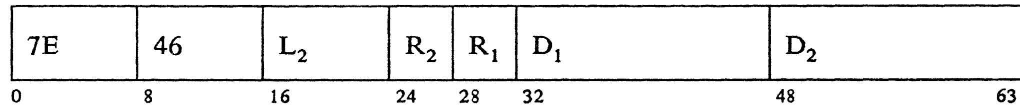
MVFXDR



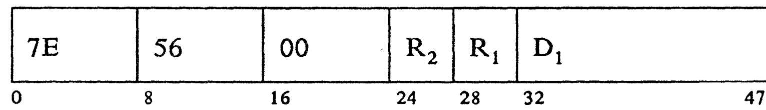
MVFXDR



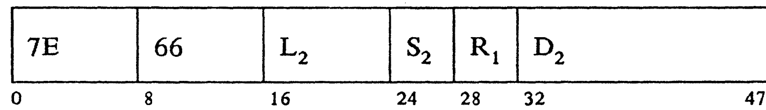
MVFXDR



MVFXDR



MVFXDR



MVFXDR

7E	76	L_2	R_2	R_1	D_2
0	8	16	24	28	32 47

MVFXDR

7E	86	00	R_2	R_1
0	8	16	24	28 31

MVSEG

28	S_1	S_2
0	8	12 15

MVSEGR

61	S_1	S_2
0	8	12 15

PAKFLD

2F	S_1	S_2	L_2	D_2
0	8	12	16	20 31

PAKFLD

77	32	0	S_1	S_2	L_2	D_2
0	8	16	20	24	28	32 47

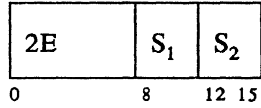
PAKFLD

77	42	0	S_1	R_2	0
0	8	16	20	24	28 31

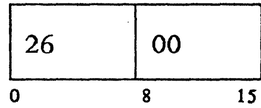
PAKFLD

77	52	0	S_1	R_2	L_2	D_2
0	8	16	20	24	28	32 47

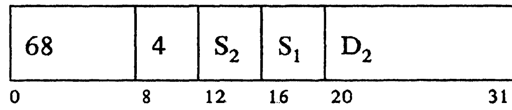
PAKSEG



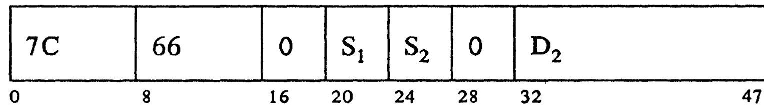
PAUSE



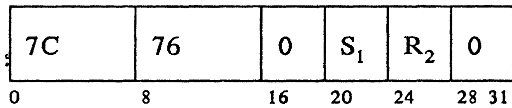
SCALE



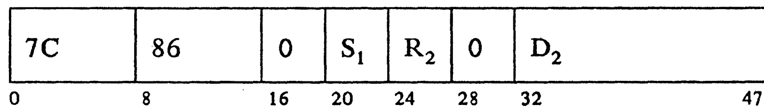
SCALE



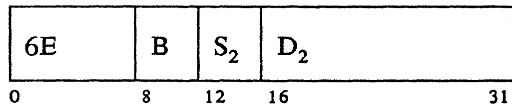
SCALE



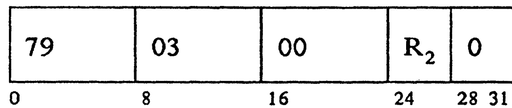
SCALE



SCRPAD



SCRPAD



SCRPAD

79	13	00	R_2	0	D_2
0	8	16	24	28	32 47

SEGALLOC

7A	4D	*	S_2	0	D_2
0	8	16	24	28	32 47

* = 02 if the WAIT Parameter = Y
0A if the WAIT Parameter = N

SEGALLOC

7A	5D	*	R_2	0
0	8	16	24	28 31

* = 02 if the WAIT Parameter = Y
0A if the WAIT Parameter = N

SEGALLOC

7A	6D	*	R_2	0	D_2
0	8	16	24	28	32 47

* = 02 if the WAIT Parameter = Y (default)
0A if the WAIT Parameter = N

SEGCOPY

17	S_1	S_2	*	D_2
0	8	12	16	20 31

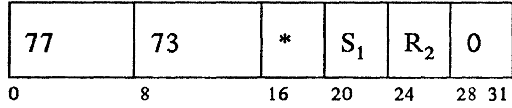
* Mask – 0 = copy to S_1; 1 = copy from S_1

SEGCOPY

77	63	*	S_1	S_2	0	D_2
0	8	16	20	24	28	32 47

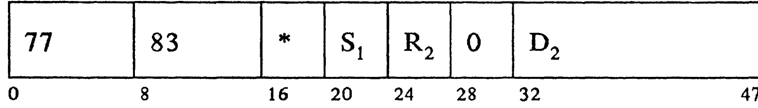
*Mask – 0 = copy to S_1; 1 = copy from S_1

SEGCOPY



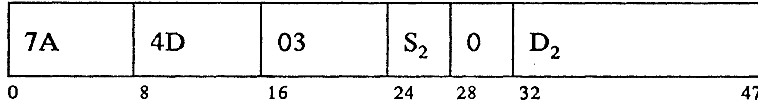
*Mask — 0 = copy to S_1; 1 = copy from S_1

SEGCOPY

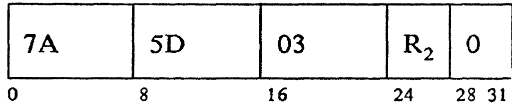


*Mask — 0 = copy to S_1; 1 = copy from S_1

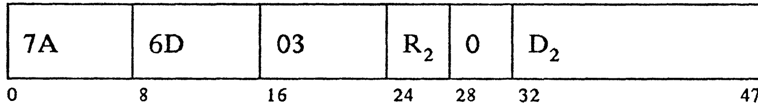
SEGFREE



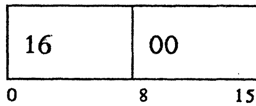
SEGFREE



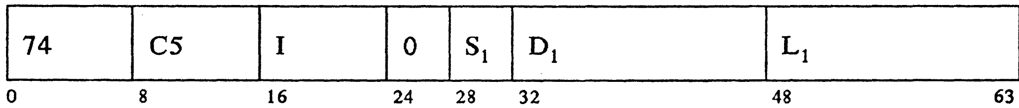
SEGFREE



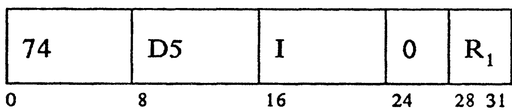
SELECT



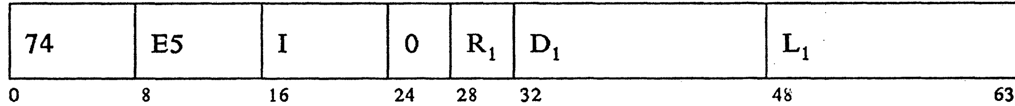
SETFLDI



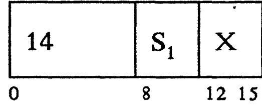
SETFLDI



SETFLDI

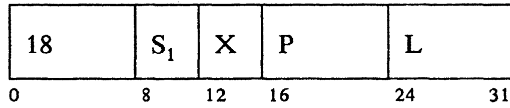


SETFPL



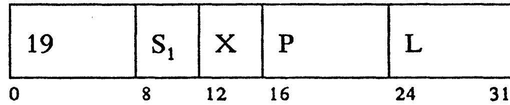
The above instruction format is generated if neither the primary field pointer (PFP) nor the field length indicator (FLI) is numeric.

SETFPL



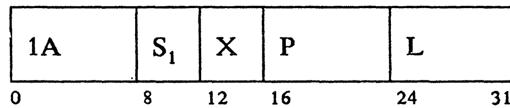
The above instruction format is generated if neither the primary field pointer (PFP) nor the field length indicator (FLI) is specified with register notation.

SETFPL



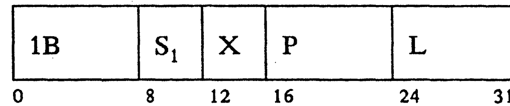
The above instruction format is generated if the primary field pointer (PFP) is not specified with register notation, and the field length indicator (FLI) is specified with register notation.

SETFPL



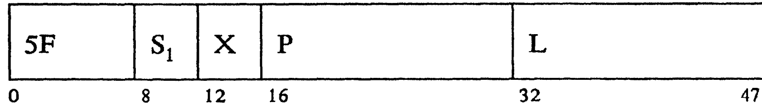
The above instruction format is generated if the primary field pointer (PFP) is specified with register notation, and the field length indicator (FLI) is not specified with register notation.

SETFPL

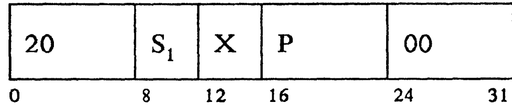


The above instruction format is generated if both the primary field pointer (PFP) and the field length indicator (FLI) are specified with register notation.

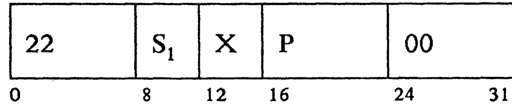
SETFPL



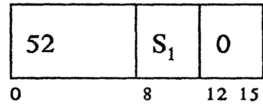
SETSFP



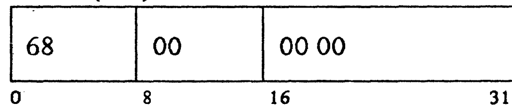
SETSFP



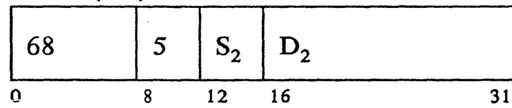
SETSFP



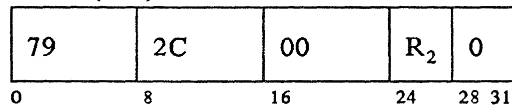
SETX (Off)



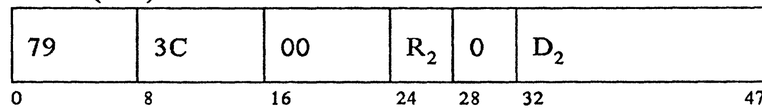
SETX (On)



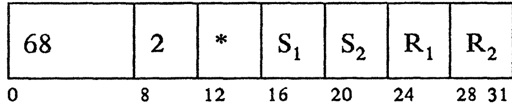
SETX (ON)



SETX (ON)

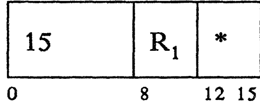


SETXREG



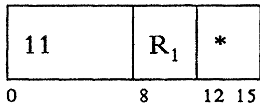
*Mask – 0 = no operands; 1 = operand 2 only; 2 = operand 1 only; 3 = both operands.

SHIFTL



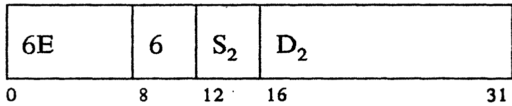
*Count value = 0-15 for shift positions 1-16

SHIFTR

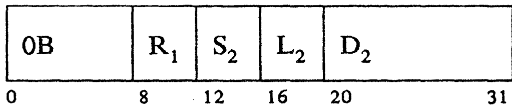


*Count value = 0-15 for shift positions 1-16

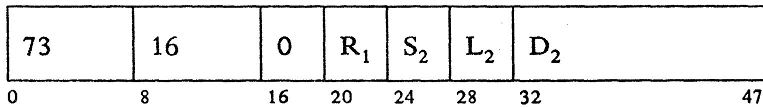
STATS



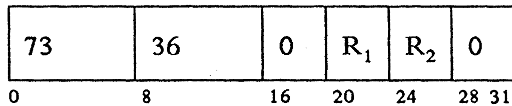
STFLD



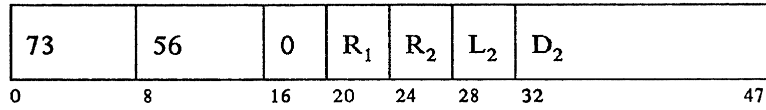
STFLD



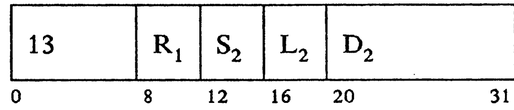
STFLD



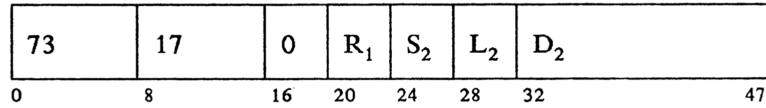
STFLD



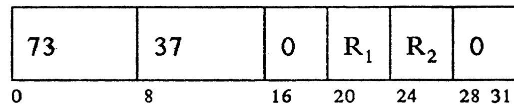
STFLDC



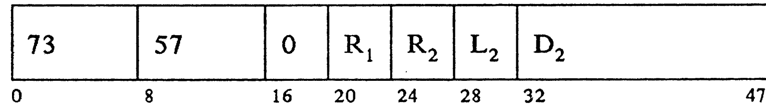
STFLDC



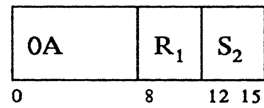
STFLDC



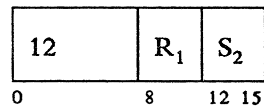
STFLDC



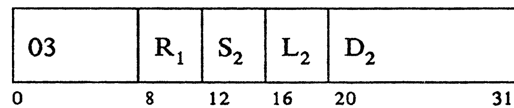
STSEG



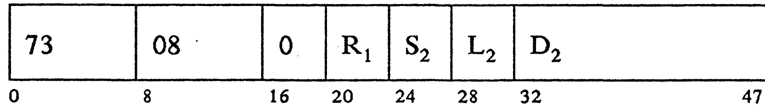
STSEGC



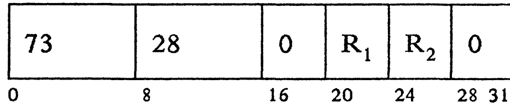
SUBFLD



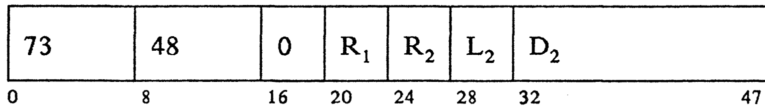
SUBFLD



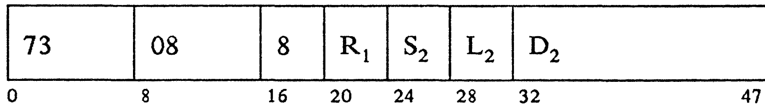
SUBFLD



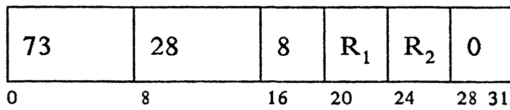
SUBFLD



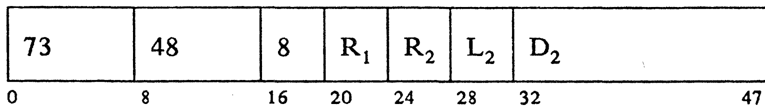
SUBFLDL



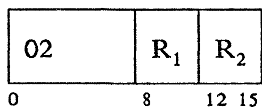
SUBFLDL



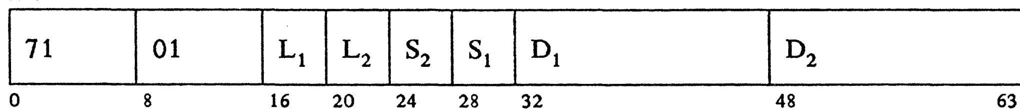
SUBFLDL



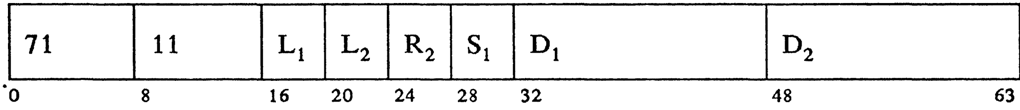
SUBREG



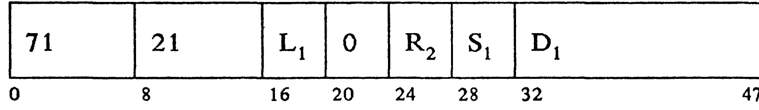
SUBZ



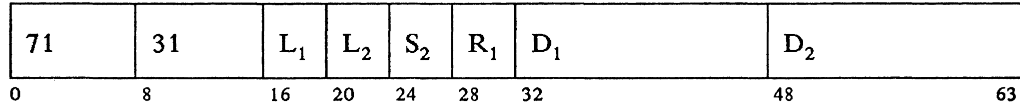
SUBZ



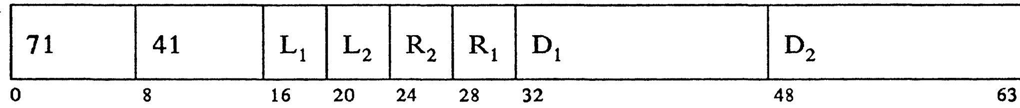
SUBZ



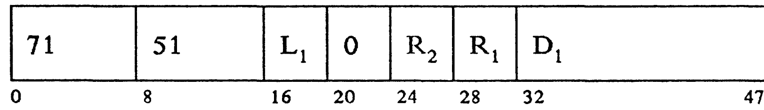
SUBZ



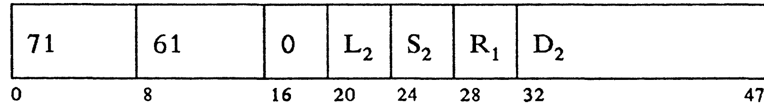
SUBZ



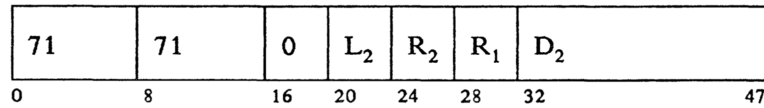
SUBZ



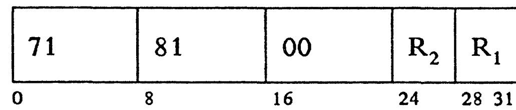
SUBZ



SUBZ



SUBZ



TESTX

68	3	S_2	D_2
0	8	12	16 31

TESTX

79	0B	00	R_2	0
0	8	16	24	28 31

TESTX

79	1B	00	R_2	0	D_2
0	8	16	24	28	32 47

TSTMSK

55	S_1	S_2	L_2	D_1	D_2
0	8	12	16	24	36 47

TSTMSK

7F	01	L_2	S_2	S_1	D_1	D_2
0	8	16	24	28	32	48 63

TSTMSK

7F	11	L_2	R_2	S_1	D_1	D_2
0	8	16	24	28	32	48 63

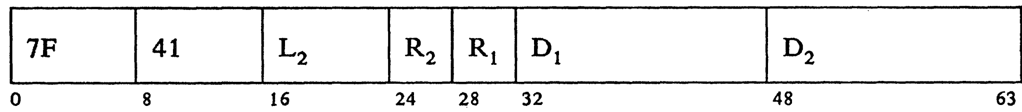
TSTMSK

7F	21	00	R_2	S_1	D_1
0	8	16	24	28	32 47

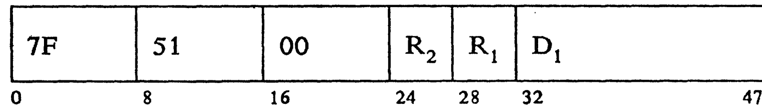
TSTMSK

7F	31	L_2	S_2	R_1	D_1	D_2
0	8	16	24	28	32	48 63

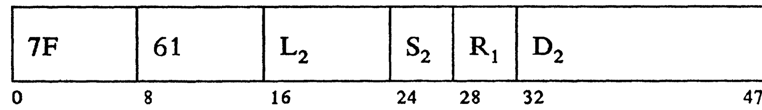
TSTMSK



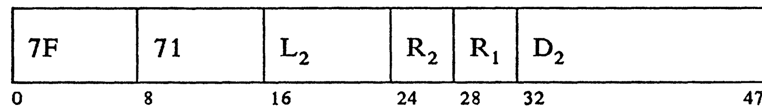
TSTMSK



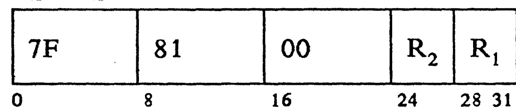
TSTMSK



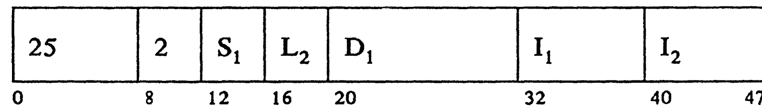
TSTMSK



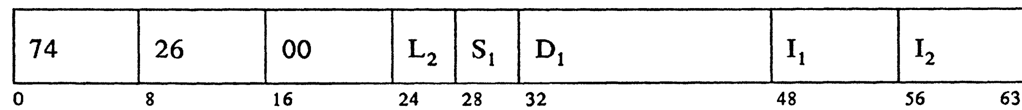
TSTMSK



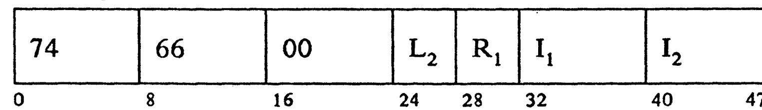
TSTMSKI



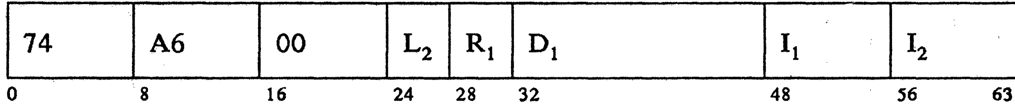
TSTMSKI



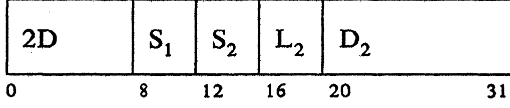
TSTMSKI



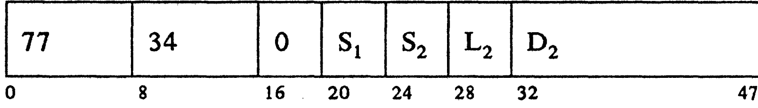
TSTMSKI



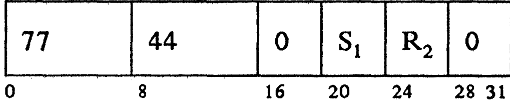
UPKFLD



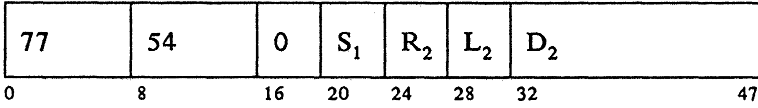
UPKFLD



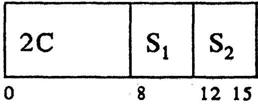
UPKFLD



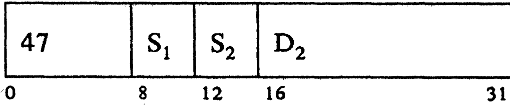
UPKFLD



UPKSEG



VERIFY



VERIFY

7C	97	0	S_1	R_2	0
0	8	16	20	24	28 31

VERIFY

7C	A7	0	S_1	R_2	0	D_2
0	8	16	20	24	28	32 47

VIEW

79	0E	00	R_2	0
0	8	16	24	28 31

VIEW

79	1E	00	R_2	0	D_2
0	8	16	24	28	32 47

VIEW

79	4E	00	S_2	0	D_2
0	8	16	24	28	32 47

Appendix B. COPY Files

This appendix gives detailed listings of the system definitions that are appropriate for this book.

Copy files may be included in your program by coding a COPY instruction specifying one of the copy file names in this appendix. For example:

```
          COPY   DEFCPL
```

You *must* code either an EQUATE or an LDSECT instruction *before* the COPY instruction to define a segment or register number for the following copy files:

- DEFCPL
- DEFDCP
- DEFELP
- DEFESP
- DEFFAP
- DEFINT
- DEFMER
- DEFSCA
- DEFSCP
- DEFSKP
- DEFSOR
- DEFTRP
- DEFTRT
- DEFTSX
- DEFVUE

If you code an EQUATE instruction, for example:

```
DEFCPLS   EQUATE n
```

then the copy file will contain a series of DEFLD instructions. The number that you specify in the EQUATE instruction will become the segment number (the first operand) of each DEFLD.

Note: The segment number must be equated to a specific label; these labels are identified for each copy file.

If you code an LDSECT instruction and specify the BASE= operand, for example:

```
          LDSECT BASE=n
          COPY   DEFCPL
          LEND
```

then the copy file will contain a series of DEFRF instructions. The number that you specify in the BASE= operand of the LDSECT instruction will become the register number (the first operand) of each DEFRF.

If you code an LDSECT and the BASE= operand before the COPY DEFAPB instruction, then the DEFAPB copy file will contain DEFRF instructions and the register number will be as specified by the BASE= operand. Otherwise the DEFAPB copy file will contain DEFLD instructions and the segment number will be 14.

You do not need to code anything before the COPY instructions for the DEFREG and DEFSEG copy files because they contain only EQUATE instructions.

The following copy files always become part of the segment specified:

Copy File	***Segment***
DEFAPB	14
DEFGMS	15
DEFRGS	0
DEFSMS	1

DEFAPB

DEFAPB

```
APBRLD   DEFxx    0,2         APB ROOT LENGTH OR ROOT CONSTANT LEN
APBXIT   DEFxx    ,2          LEXIT INSTRUCTION
APBPID   DEFxx    ,8          APB NAME
APBGID   DEFxx    ,6          APB GENERATION DATE
APBVER   DEFxx    ,1          APB VERSION NUMBER
APBLVL   DEFxx    ,1          APB LEVEL NUMBER
         DEFxx    ,4          RESERVED
APBLTH   DEFxx    ,2          LENGTH OF SEGMENT 14
APBROD   DEFxx    ,2          POINTER TO RESIDENT OVERLAY
                                DIRECTORY

APBFLG   DEFxx    ,1          FLAG BYTE:
APBMVO   DEFxx    APBFLG,1      OPTION FLAG
APBMVOM  EQUATE    X'01'            OPTION FLAG MASK

APBSPR   DEFxx    ,1          RESERVED
APBSUE   DEFxx    ,2          STARTUP       ENTRY POINT
APBAPC   DEFxx    ,2          APCALL        ENTRY POINT
APBPCE   DEFxx    ,2          PROGRAM CHECK ENTRY POINT
         DEFxx    ,2          RESERVED
APBAPI   DEFxx    ,2          ASYNCHRONOUS  ENTRY POINT - POST
APBATE   DEFxx    ,2          ASYNCHRONOUS  ENTRY POINT - TERMINAL
APBACE   DEFxx    ,2          ASYNCHRONOUS  ENTRY POINT - CPU
APBASE   DEFxx    ,2          ASYNCHRONOUS  ENTRY POINT - STATION
APBTME   DEFxx    ,2          ASYNCHRONOUS  ENTRY POINT - TIMER
APBALA   DEFxx    ,2          ASYNCHRONOUS  ENTRY POINT - ALA
APBPCL   DEFxx    ,2          RESERVED
APBIVN   DEFxx    ,2          1ST HWD OF INSTR SECTION ADR
APBIAD   DEFxx    ,2          2ND HWD OF INSTR SECTION ADR
APBSDT   DEFxx    ,2          SEGMENT DEFINITION TABLE ADDR
APBDEL   DEFxx    ,2          DELIMITER TABLE ADDRESS
         DEFxx    ,10         RESERVED

APBFUI   DEFxx    ,0          FIRST USER CONSTANT OR INSTRUCTION
```

Segment 14 Fields

Segment 14 contains the following fixed fields generated by the BEGIN instruction of the application program:

Length of Root Section (APBRLD): The length of the root section of the controller application program.

Exit Instruction (APBXIT): An LEXIT instruction executed by the controller when the logical work station must be forced to give up control.

Application Program Name (APBPID): An 8-byte character field containing the application program's name.

Month, Day, and Year Assembled (APBGID): A 6-byte character field containing the date specified in the BEGIN instruction (the DATE operand).

Version Number (APBVER): A 1-byte character field containing the version number of the application program.

Release Level (APBLVL): The application program assembly release level.

Total Length of Program (APBLTH): The length of the largest aggregate of root section and overlay sections.

Resident Overlay Directory Pointer (APBROD): The address of the beginning of the resident overlay directory.

Flag Byte (APBFLG): Reserved for system use.

Startup Entry Point (APBSUE): The address of the startup entry point, if specified.

Calling Entry Point (APBAPC): The APCALL entry point address, if specified.

Program Check Entry Point (APBPCE): The address of the program check entry point, if specified.

Asynchronous Post Entry Point (APBAPI): The address of the application program entry point to be used by LPOST, if specified.

Asynchronous Terminal Entry Point (APBATE): The address of the terminal entry point, if specified.

Asynchronous CPU Entry Point (APBACE): The address of the host processor entry point, if specified.

Asynchronous Station Entry Point (APBASE): The address of the station entry point, if specified.

Asynchronous Timer Entry Point (APBTME): The address of the application program entry point to be used when a work station is dispatched because the work station Segment 1 timer value (SMSTMR) is less than or equal to the value in the controller's timer (contained in the GMSTMR field in Segment 15).

Optional Host Link Entry Point (APBALA): Address of the feature host processor (non-SNA/SDLC) link entry point.

Instruction Section Address (APBIVN/APBIAD): A 4-byte field containing the application root section address.

Segment Definition Table (APBSDT): Displacement of the SDT, created by DEFSTOR, in Segment 14.

Address of Delimiter Table (APBDEL): The address of the delimiter table, if specified.

Application Program (APBFUI): The first constant or instruction in the application program.

DEFCPL

Equate DEFCPLS to a segment number.

DEFCPL

```
CPLPAR   DEFxx    s,19          COMPRESS/COMPACT PARMETER LIST
CPLINS   DEFxx     CPLPAR,2     SEGMENT CONTAINING INPUT AREA
CPLIND   DEFxx    s,2           DISPLACEMENT TO INPUT AREA
CPLINL   DEFxx    s,2           LENGTH OF INPUT AREA
CPLOUS   DEFxx    s,2           SEGMENT CONTAINING OUTPUT ARE
CPLOUD   DEFxx    s,2           DISPLACEMENT TO OUTPUT AREA
CPLOUL   DEFxx    s,2           LENGTH OF OUTPUT AREA
CPLPRI   DEFxx    s,1           PRIME COMPRESSION CHARACTER
CPLFLG   DEFxx    s,1           INPUT FLAG BYTE
CPLFCT   EQUATE   X'80'                PERFORM COMPACTION
CPLFCR   EQUATE   X'40'                PERFORM COMPRESSION
CPLTBS   DEFxx    s,2           SEG CONTAINS COMPACT TABLE IN
CPLTBD   DEFxx    s,2           DISPLACEMENT TO TABLE AREA
CPLTST   DEFxx    s,1           TERMINATION STATUS
CPLTOV   EQUATE   X'80'                OUTPUT OVERFLOW
CPLTIL   EQUATE   X'40'                INITIAL INPUT LENGTH IS
                                         ZERO
```

DEFDCP

Equate DEFDCPS to a segment number.

DEFDCP

```
DCPPAR   DEFxx    s,19          DECOMPRESS/DECOMPACT PARAMETER
                                  LIST
DCPINS   DEFxx     DCPPAR,2     SEGMENT CONTAINING INPUT AREA
DCPIND   DEFxx    s,2           DISPLACEMENT TO INPUT AREA
DCPINL   DEFxx    s,2           LENGTH OF INPUT AREA
DCPOUS   DEFxx    s,2           SEGMENT CONTAINING OUTPUT AREA
DCPOUD   DEFxx    s,2           DISPLACEMENT TO OUTPUT AREA
DCPOUL   DEFxx    s,2           LENGTH OF OUTPUT AREA
DCPPRI   DEFxx    s,1           PRIME COMPRESSION CHARACTER
DCPFLG   DEFxx    s,1           INPUT FLAG BYTE
DCPFCT   EQUATE   X'80'                INPUT DATA IN COMPACT
                                       CODE
DCPTBS   DEFxx    s,2           SEGMENT CONTAINING DEC TABLE
DCPTBD   DEFxx    s,2           DISP TO DECOMPACTION TABLE
DCPTST   DEFxx    s,1           TERMINATION STATUS
DCPTOV   EQUATE   X'80'                OUTPUT OVERFLOW
DCPTIV   EQUATE   X'40'                INPUT OVERFLOW
DCPTCE   EQUATE   X'20'                COMPACT CODE INPUT
DCPTSL   EQUATE   X'10'                SCB COUNT FIELD IS ZERO
DCPTIL   EQUATE   X'08'                INITIAL INPUT LENGTH IS
                                         ZERO
```

DEFELP

Equate DEFELPS to a segment number.

DEFELP

```
ELPARM   DEFxx   s,14         ENTIRE PARAMETER LIST
ELPOVN   DEFxx    ELPARM,8    OVERLAY NAME
ELPFLG   DEFxx   s,1          SECTION LOAD FLAGS
ELPCSF   EQUATE    X'80'                 LOAD CONSTANT SECTION
ELPISF   EQUATE    X'40'                 LOAD INSTRUCTION SECTION
ELPSEG   DEFxx   s,1          CONSTANT SECTION SEGMENT NUMBER
ELPCLA   DEFxx   s,2          CONSTANT SECTION LOAD ADDRESS
ELPILA   DEFxx   s,2          INSTRUCTION SECTION LOAD ADDRESS
```

DEFESP

Equate DEFESPS to a segment number.

DEFESP

```
ESPREQ   DEFxx    ,1          REQUEST CODE
ESPFLG   DEFxx    ,1          OPTION FLAGS
ESPRST   EQUATE    X'80'               RESET COUNTER AFTER READOUT
ESPNXT   EQUATE    X'40'               REQUEST NEXT COUNTERS DATA
ESPEID   DEFxx    ,2          EXTENDED COUNTER ID
ESPDID   DEFxx    ,2          DEVICE ID
ESPTOT   DEFxx    ,6          TOTAL BYTES
ESPERR   DEFxx    ,4          ERROR BYTES
ESPNDV   DEFxx    ,1          NUMBER OF DEVICES ASSIGNED
ESPSID   DEFxx    ESPEID,2    DEVICE STAT CTR ID
ESPTYP   DEFxx    ,1          DEVICE TYPE CODE
ESPFEA   DEFxx    ,1          FEATURE FLAGS
ESPSTA   DEFxx    ,1          ASSIGNED STATION ID
ESPNCT   DEFxx    ,1          NUMBER OF COUNTERS
ESPSCTR  DEFxx    ,32         BYTE COUNTERS

*
* * * *  DEFINITION OF FEATURE FLAGS IN ESPFEA
*
* LOOP (DEVICE TYPE X'80'
*
ESPLPCLK EQUATE    X'80'        CLOCKING LOOP
ESPLP06  EQUATE    X'08'        600 BPS SPEED
ESPLP12  EQUATE    X'04'        1200 BPS SPEED
ESPLP24  EQUATE    X'02'        2400 BPS SPEED
ESPLP48  EQUATE    X'01'        4800 BPS SPEED

*
* COMMON FOR ALL LOOP ATTACHED DEVICES
*
ESPFFRA  EQUATE    X'08'        READ OPERATION DEVICE
ESPFFWA  EQUATE    X'04'        WRITE OPERATION DEVICE
ESPFFSA  EQUATE    X'02'        SHARED ADDRESS SLOT DEVICE
ESPFFSO  EQUATE    X'01'        SHARED OPERATOR DEVICE

*
* FLAGS THAT ARE DEVICE DEPENDENT
*
* 3604 DISPLAY
ESPDSCUR EQUATE    X'80'        CURSOR SET ON AT DEVICE INITIALIZATION

*
* 3604 KEYBOARD
ESPKBCUR EQUATE    X'80'        CURSOR IS NOT SET ON/OFF AT READ TIME
ESPKBERT EQUATE    X'40'        ERTSL OPTION
ESPKBPIN EQUATE    X'10'        PIN PAD AVAILABLE AND USED
```

```
*
* 3610/3612 DOCUMENT PRINTER
ESPPRPFT EQUATE    X'20'        PIN FEED TRACTOR
ESPPRACT EQUATE    X'10'        AUTOMATIC CUT FORM MODE

*
* 3618 ADMINISTRATIVE PRINTER
ESPADEPL EQUATE    X'80'        EXPANDED PRINT LINE
ESPADDFF EQUATE    X'20'        DUAL FORMS FEED
*
```

DEFFAP

Equate DEFFAPS to a segment number.

DEFFAP

```
FAPARM   DEFxx    ,49          ENTIRE PARAMETER LIST
FAPAPN   DEFxx    FAPARM,8     AP NAME
FAPFLG   DEFxx    ,1           REQUEST/RESPONSE FLAGS
FAPCLR   EQUATE    X'80'                 INPUT: READ=0 / READ,
                                           RESET=1
FAPTRN   EQUATE    X'40'                 OUTPUT: RESIDENT=0 /
                                           TRANSIENT=1
FAPFCN   DEFxx    ,1           FUNCTION CODE
FAPAPCNT EQUATE   X'00'          GET AP USAGE COUNTS
FAPASTOR EQUATE   X'01'          GET AP STORAGE USAGE
FAPSSTOR EQUATE   X'02'          GET STATION STORAGE USAGE
FAPCSTOR EQUATE   X'03'          GET POOL STORAGE DATA
FAPSTATS EQUATE   X'04'          GET POOL STATISTICS
FAPCAL   DEFxx    ,2           NUMBER OF CALLS MADE
FAPLOD   DEFxx    ,2           NUMBER OF LOADS FROM DISKETTE
FAPWAT   DEFxx    ,2           NUMBER OF WAITS FOR BUFFER SPACE
FAPCSH   DEFxx    FAPCAL,1     POOL ID/STATION ID
         DEFxx    ,1           RESERVED
FAPTTL   DEFxx    ,3           POOL SIZE IN BYTES
FAPSTOR  DEFxx    ,3           STORAGE OWNED BY STATION OR AP
*                              OR IN USE BY A POOL
FAPSTQ   DEFxx    FAPSTOR,1    CURRENT LENGTH OF WAIT QUEUE
FAPSTN   DEFxx    ,1           STATION CURRENTLY OWNING THIS POOL
FAPCBY   DEFxx    ,2           COMBINE INITIATE THRESHOLD
FAPCBN   DEFxx    ,2           COMBINE INHIBIT THRESHOLD
FAPCUR   DEFxx    ,3           CURRENT BYTES IN USE
FAPSMAX  DEFxx    ,3           MAXIMUM BYTES IN USE
FAPSAPC  DEFxx    ,2           COUNT OF APCALLS/DTACCESSES
FAPSRCL  DEFxx    ,2           TOTAL NUMBER OF RECLAIMS
FAPSREQ  DEFxx    ,2           TOTAL NUMBER OF REQUESTS
FAPSCMB  DEFxx    ,2           NUMBER OF SEGMENT COMBINES
FAPSDEC  DEFxx    ,2           NUMBER OF SEGMENT DECOMPOSES
FAPSSCN  DEFxx    ,2           AVG SCAN TO SATISFY A REQUEST
FAPSMOV  DEFxx    ,2           AVG # BLOCK MOVES IN SHORT GARBAGE
                                 COLL
FAPSLNG  DEFxx    ,2           AVG SUCCESS RATIO FOR LONG GARBAGE
                                 COLL
FAPSWTC  DEFxx    ,2           COUNT OF NUMBER OF POOL WAITS
FAPSWTS  DEFxx    ,2           AVG # SECONDS WAITING FOR POOL
FAPSWTQ  DEFxx    ,2           AVG LENGTH OF WAIT QUEUE
*                                 AMOUNT IN USE BY POOL
```

DEFGMS

DEFGMS

```
GMSTMR   DEFLD     15,0,4       GLOBAL SEGMENT 15 TIMER
GMSHTM   DEFLD     15,0,2       HIGH ORDER 2 BYTES OF TIMER
GMSLTM   DEFLD     15,,2        LOW ORDER 2 BYTES OF TIMER
GMSWTM   DEFLD     15,,4        CONTROLLER WAIT TIME (IN SECONDS)
GMSHRT   DEFLD     15,,2        HIGH RESOLUTION TIMER
```

```
GMSMFS   DEFLD    15,,2        MACHINE FEATURE SWITCHES:
GMSMTDM  EQUATE   X'2000'        TWO DISKETTE ADAPTERS
GMSMDCM  EQUATE   X'1000'        DCA ADAPTER
GMSMBLM  EQUATE   X'0700'        MASK TO EXTRACT NUM OF B-LOOPS
GMSMHLM  EQUATE   X'00F0'        MASK TO EXTRACT HOST LINK TYPE:
GMSMH0M  EQUATE   X'0000'          NO HOST LINK
GMSMH1M  EQUATE   X'0010'          HPCA - X.21 SWITCHED
GMSMH2M  EQUATE   X'0020'          HPCA - EIA
GMSMH3M  EQUATE   X'0030'          HPCA - MULTI USE LOOP
GMSMH5M  EQUATE   X'0050'          CCA  - EIA
GMSMH6M  EQUATE   X'0060'          HPCA - X.25 EIA
GMSMH7M  EQUATE   X'0070'          HPCA - X.21 LEASED
GMSMH8M  EQUATE   X'0080'          HPCA - X.25 X.21

GMSDSM   DEFLD    15,,1        RESERVED FOR SYSTEM USE
GMSMSZ   DEFLD    15,,1        NUMBER OF 64K SECTIONS OF MEMORY

GMSFTR   DEFLD    15,,2        OPTIONAL MODULES LOADED FLAGS:
GMSFEDM  EQUATE   X'8000'        ENCODE/DECODE (AET)
GMSFSMM  EQUATE   X'2000'        LSORT/LMERGE
GMSFSDM  EQUATE   X'1000'        SETDSKT
GMSF6AM  EQUATE   X'0800'        STARTER ADAPTER DIAGS MASK
GMSFSLM  EQUATE   X'0400'        ALTERNATE LINE MASK
GMSF68M  EQUATE   X'0200'        SCALE/SETX.../TESTX
GMSFDEM  EQUATE   X'0100'        ENCODE/DECODE (DES)
GMSFMTM  EQUATE   X'0080'        DISKETTE MULTI-BLOCK I/O MASK
GMSF20M  EQUATE   X'0040'        THIS FLAG IS ALWAYS SET, AS
                                 LCHAP IS IN THE BASE

GMSF25M  EQUATE   X'0020'        STATS
GMSF21M  EQUATE   X'0010'        LTRT
GMSF6DM  EQUATE   X'0040'        LEFT FOR IR COMPATIBILITY
GMSF26M  EQUATE   X'0008'        DECOMP/DECOMPTB
GMSF70M  EQUATE   X'0004'        DATA STREAM MANAGEMENT
GMSF24M  EQUATE   X'0002'        LSEEKP
GMSF27M  EQUATE   X'0001'        COMP/COMPTB

GMSFT2   DEFLD    15,,2        OPTIONAL MODULES LOADED FLAGS:
GMSF2AM  EQUATE   X'8000'        SCRPAD
GMSF2CM  EQUATE   X'2000'        INTMR
GMSF2DM  EQUATE   X'1000'        THIS FLAG IS ALWAYS SET, AS
*                                LLOAD WITH PARM=EXP IS IN BASE
GMSF28M  EQUATE   X'0800'        SECURITY INSTRUCTIONS
GMSF31M  EQUATE   X'0400'        ZONED DECIMAL INSTRUCTIONS
GMSF32M  EQUATE   X'0200'        LTIMEV INSTRUCTION
GMSF40M  EQUATE   X'0100'        FORMDKT
GMSF41M  EQUATE   X'0080'        COMPDKT
GMSF29M  EQUATE   X'0040'        TRANSIENT APCALL/APRETURN
GMSF34M  EQUATE   X'0020'        LCONVERT/CRETN
GMSF36M  EQUATE   X'0010'        DTACCESS/SEGALLOC
GMSF42M  EQUATE   X'0008'        DPOOL
GMSF2FM  EQUATE   X'0004'        FORMDSK
GMSFB7M  EQUATE   X'0002'        DISK MULTI-BLOCK I/O MASK
         DEFLD    15,,2        RESERVED

GMSCUA   DEFLD    15,,1        CONTROL UNIT ADDRESS
GMSLID   DEFLD    15,,1        COMMUNICATION LINK ID:
GMSL48M  EQUATE   X'01'          4800 BPS
GMSL96M  EQUATE   X'02'          9600 BPS
GMSL02M  EQUATE   X'02'          4502 - SNA/SDLC
GMSL05M  EQUATE   X'05'          1422 - BSC
GMSL07M  EQUATE   X'07'          5656 - X.21
GMSL08M  EQUATE   X'08'          4850 - MULTI-USE
GMSL09M  EQUATE   X'09'          8V0134 - X.25
```

```
GMSRES1  DEFLD     15,,2          RESERVED
GMSBLN   DEFLD     15,,2          XID BLOCK NUMBER
GMSMOD   DEFLD     15,,1          INPUT MSG ROUTING CONTROL
GMSDEF   DEFLD     15,,1          DEFAULT STB ID IF DEFAULT USE
GMSSDI   DEFLD     15,GMSDEF,1    CLEAR STB ID DISP IN MSG
GMSTAB   DEFLD     15,,2          POINTER TO ROUTING TABLE
GMSSPR   DEFLD     15,,1          RESERVED
GMSLOP   DEFLD     15,,1          SUMMARY LOOP OUTAGE MAP

GMSIND   DEFLD     15,,1          GLOBAL INDICATOR BYTE:
GMSILDM  EQUATE    X'80'            LOSS OF CONTACT MASK
GMSILRM  EQUATE    X'40'            LINK DOWN MASK
GMSIWSM  EQUATE    X'20'            WARM START MASK
GMSIDSM  EQUATE    X'10'            2-SIDED DISKETTE ON PRIMARY
                                      DRIVE
GMSISMM  EQUATE    X'04'            SYS MONITOR LOGGED ON AT 3604
GMSIDCM  EQUATE    X'02'            DISK CREATE IN PROGRESS FLAG
GMSIMAM  EQUATE    X'01'            MSG REQ ACTION WRITTEN TO LOG

GMSFLG   DEFLD     15,,1          FLAG BYTE:
GMSFPCM  EQUATE    X'80'            MON PROC CMD IN PROG I/P MODE
GMSID2M  EQUATE    X'40'            2-SIDED DISKETTE ON SECONDARY
                                      DRIVE
GMSD2PM  EQUATE    X'20'            SECONDARY DISK DRIVE PRESENT
GMSCSCM  EQUATE    X'10'            CNM SPECIFIED IN CONFIGURATION
GMSAPCM  EQUATE    X'08'            AP'S TO DISK REQ'TD IN CONFIG-
                                      URATION

         DEFLD     15,,2          RESERVED
GMSBSN   DEFLD     15,,2          CURRENT TF BLOCK NUMBER ON IPL
                                    DRIVE
         DEFLD     15,,2          RESERVED
GMSBSA   DEFLD     15,,2          CURRENT TF BLOCK NUMBER ON 2ND
                                    DRIVE
GMSBSD   DEFLD     15,,4          CURRENT TF BLOCK NUMBER ON 1ST
                                    DISK
GMSBSD2  DEFLD     15,,4          CURRENT TF BLOCK NUMBER ON 2ND
                                    DISK
GMSPRI   DEFLD     15,,1          PRIORITY DISPATCH (FLAG AND
                                    TABLE NUM)

GMSTYP   DEFLD     15,,1          DISKETTE TYPE MOUNTED FLAGS
GMSITYM  EQUATE    X'F0'            MASK TO EXTRACT IPL DRIVE FLAGS
GMSID1M  EQUATE    X'10'              TYPE 1  DKT  ON IPL DRIVE
GMSIDT2M EQUATE    X'20'              TYPE 2  DKT  ON IPL DRIVE
GMSID2DM EQUATE    X'30'              TYPE 2D DKT  ON IPL DRIVE

GMSSTYM  EQUATE    X'0F'            MASK TO EXTRACT 2ND DRIVE FLAGS
GMSST1M  EQUATE    X'01'              TYPE 1  DKT  ON 2ND DRIVE
GMSST2M  EQUATE    X'02'              TYPE 2  DKT  ON 2ND DRIVE
GMSST2DM EQUATE    X'03'              TYPE 2D DKT  ON 2ND DRIVE

GMSSID   DEFLD     15,,2          SESSION ID

GMSDKTP  DEFLD     15,,1          DISK DRIVE FLAGS
GMSDSK1  EQUATE    X'F0'          1ST DISK DRIVE CAPACITY MASK
GMSD1LM  EQUATE    X'20'          1ST DISK DRIVE IS LARGE CAPACITY
GMSD1SM  EQUATE    X'10'          1ST DISK DRIVE IS SMALL CAPACITY
GMSDSK2  EQUATE    X'0F'          2ND DISK DRIVE CAPACITY MASK
GMSD2LM  EQUATE    X'02'          2ND DISK DRIVE IS LARGE CAPACITY
         DEFLD     15,,1          RESERVED
GMSCSN   DEFLD     15,,3          CONTROLLER SERIAL NUMBER

GMSMFS3  DEFLD     15,,1          ALA LINE
GMSAL1M  EQUATE    X'F0'          MASK TO EXTRACT ALA LINE :
GMSM1SM  EQUATE    X'20'          ALA  - SNAP (HPCA) ADAPTER
GMSM1AM  EQUATE    X'40'          ALA  - START/STOP (CCA) ADAPTER
```

```
GMSMFS4  DEFLD      15,,1          RESERVED

GMSMFS5  DEFLD      15,,1          RESERVED

GMSMFS6  DEFLD      15,,1          RESERVED

GMSIND2  DEFLD      15,,1        GLOBAL INDICATOR BYTE 2:
GMSILD2M EQUATE      X'80'         LINK2 LOSS-OF-CONTACT MASK
GMSILR2M EQUATE      X'40'         LINK2 LINK DOWN MASK
GMSCUA2  DEFLD      15,,1          LINK2 CONTROL UNIT ADDRESS
GMSLID2  DEFLD      15,,1          LINK2 COMMUNICATIONS LINK ID

GMSRES   DEFLD      15,,12         RESERVED

GMSFUF   DEFLD      15,,0          FIRST USER FIELD
```

Segment 15 Fields

The following fields are in the fixed area of Segment 15.

Controller Timer (GMSTMR): A timer with a resolution of 1 second. Any value set in this field is increased by 1 each second. Controller application programs can set and reset this field.

Controller Wait Time (GMSWTM): The number of seconds the controller is waiting for work. Each time the controller accumulates 1 second of 'waiting for work', it adds 1 to the value in this field.

High Resolution Timer (GMSHRT): A timer whose value is increased by 1 each 64 milliseconds.

Machine Feature Switches (GMSMFS): Indicate the controller configuration, adapters, host link, and so on. Bit patterns are as follows:

Bit Position	Meaning
xx1x xxxx xxxx xxxx	Two diskette adapters present
xxx1 xxxx xxxx xxxx	DCA adapter
xxxx xNNN xxxx xxxx	NNN = Number of B-Loops
xxxx xxxx 0000 xxxx	No host link
xxxx xxxx 0001 xxxx	Host link is HPCA - X21 Switched
xxxx xxxx 0010 xxxx	Host link is HPCA - EIA
xxxx xxxx 0011 xxxx	Host link is HPCA - Multi Use Loop
xxxx xxxx 0101 xxxx	Host link is CCA - EIA
xxxx xxxx 0110 xxxx	Host link is HPCA - X25
xxxx xxxx 0111 xxxx	Host link is HPCA - X21 Leased
xxxx xxxx 1000 xxxx	Host link is HPCA - X25 - X21
xxxx xxxx xxxx 1xxx	First disk drive present
xxxx xxxx xxxx x1xx	Second disk drive present

Reserved Field (GMSDSM): A 1-byte field reserved for system use.

Memory Size (GMSMSZ): The number of 64K sections of main storage in this controller.

Features Flag Fields (GMSFTR/FT2): Two 2-byte fields that have bits set to indicate the optional modules loaded in the controller.

Control Unit Address (GMSCUA): SDLC control unit address.

Communication Link ID (GMSLID): A one-byte field indicating which controller communication link access method was loaded as follows:

Hex Value:	Meaning:
02	SNA/SDLC
05	BSC3
07	X.21
08	Multiuse loop
09	X25

SNA Node Identification Block Number (GMSBLN): A 2-byte field containing the IBM-assigned SNA node identification block number.

Input Message Routing Control (GMSMOD): Reserved.

Default Station ID (GMSDEF): Reserved.

Routing Table Pointer (GMSTAB): Reserved.

Loop Outage Indicator (GMSLOP): Indicates loops that did not start successfully, or have failed after start. A bit set to 1 indicates that the corresponding loop is not operational. Bits 0 — 3 correspond to loops 1 — 4 with bit 0 = loop 1. Bits corresponding to loops specified in the configuration for this load image are initially set to 1. As a loop starts successfully, its bit is set to 0. When a running loop becomes nonoperational, its bit is set to 1. This field can be queried by the application program and messages issued to the host control operator if a loop fails to start or becomes nonoperational.

Global Indicator (GMSIND): The status of the communication link and controller communication adapter, and whether the current controller startup was a warm start or a cold start, whether a one-sided or two-sided diskette is currently mounted in the controller plus other information. Bit patterns are as follows:

Bit Position	Meaning
01xx xxxx	Link is running (adapter enabled)
11xx xxxx	Loss of contact (adapter enabled)
x0xx xxxx	Adapter disabled (contact bit ignored)
xx1x xxxx	Warm start
xx0x xxxx	Cold start
xxx1 xxxx	Two-sided diskette mounted in primary drive
xxx0 xxxx	One-sided diskette mounted in primary drive
xxxx x1xx	System monitor logged on to 4704/3604/3278
xxxx xx1x	Diskette create in progress
xxxx xxx1	Message written to log

Flag Field (GMSFLG): A 1-byte flag field:

Bit Position	Meaning
1xxx xxxx	The system monitor is processing a command issued from a controller application program.
x1xx xxxx	A 2-sided diskette is in the secondary drive.
xx1x xxxx	A secondary diskette is present in the configuration.
xxx1 xxxx	CNM is specified in the configuration.
xxxx 1xxx	User application program overlays can reside on disk.

Current Temporary File Block Number (GMSBSN): The number of the temporary file block being used for new temporary file records on the primary diskette drive.

Current Temporary File Block Sequence Number (GMSBSA): The number of the temporary file block being used for new temporary file records on the secondary diskette drive.

Current Disk Temporary File Block Number (GMSBSD): The number of the temporary file block being used for new temporary file records on the first disk.

Current Disk Temporary File Block Number (GMSBSD2): The number of the temporary file block being used for new temporary file records on the second disk.

Priority Dispatching (GMSPRI): Bits 1 — 7 contain the ID of the priority dispatching table. Bit 0, when set to 1, indicates that priority dispatching is in effect for the table whose ID is in bits 1 — 7.

Diskette Type (GMSTYP): A 1-byte field indicating the mounted diskette type.

Controller Session ID (GMSSID): The number of times a system cold start or a SETDSKT-reset-Temporary-File has been performed using a specific diskette. Each cold start of the controller increments the session ID and erases the data in the diskette Temporary File.

Disk Drive Flags (GMSDKTP): A flag byte indicating the disk 1 and 2 capacities:

Bit Position	Meaning
0010 xxxx	First disk drive is large capacity.
0001 xxxx	First disk drive is small capacity.
xxxx 0010	Second disk drive is large capacity.
xxxx 0001	Second disk drive is small capacity.

Controller Serial Number (GMSCSN): The unique identifier of the controller.

ALA Line Field (GMSMFS3): A 1-byte field describing the adapters attached to the ALA line, as follows:

Bit Position	Meaning
0000 0000	No adapter
0100 0000	ALA-Start/Stop(CCA) adapter
0010 0000	ALA-SNAP(HPCA) adapter

First User Field (GMSFUF): The first user-defined field in Segment 15.

DEFINT

Equate DEFINTS to a segment number.

DEFINT

```
INTTMR    DEFxx    s,16          INTERVAL TIMER PARMLIST
INTSID    DEFxx     INTTMR,1     WORKSTATION NUMBER
INTTNO    DEFxx    s,1           INTERVAL TIMER NUMBER
INTREQ    DEFxx    s,1           REQUEST TYPE

INTSTRTM EQUATE X'00'                  REQUEST IS START TIMER
INTSTOPM EQUATE X'01'                  REQUEST IS STOP TIMER
INTSTPRM EQUATE X'11'                  REQUEST IS STOP & RETURN
                                         INTERVAL
INTREADM EQUATE X'02'                  REQUEST IS READ RESULTS
INTRDRSM EQUATE X'03'                  REQUEST IS READ & RESET
INTACTVM EQUATE X'04'                  REQUEST IS ACTIVATE TIMING
INTDACTM EQUATE X'05'                  REQUEST IS DEACTIVATE TIMING

*
*      THIS ENDS THE BASIC PORTION OF INTERVAL TIMER PARAMETER LIST
*
*
*      THE FOLLOWING FIELD USED WITH READ, READ & RESET, AND
*          STOP & REPORT RESULTS REQUESTS
*
INTFLG    DEFxx    s,1           FLAG BYTE
          LSPACE
INTDAVTM EQUATE X'80'                  TIMER IS DEACTIVATED FLAG
INTSTARM EQUATE X'40'                  TIMER IS CURRENTLY RUNNING

*
*      THE FOLLOWING FIELD USED WITH STOP & REPORT RESULTS REQUEST
*
INTLTH    DEFxx    s,3           LENGTH OF INTERVAL
*
*      THE FOLLOWING FIELDS USED WITH READ AND READ & RESET
*        REQUESTS
INTMIN    DEFxx     INTLTH,3     SHORTEST INTERVAL
INTMAX    DEFxx    s,3           LONGEST INTERVAL
INTTOT    DEFxx    s,4           SUM OF ALL INTERVALS
INTCNT    DEFxx    s,2           NUMBER OF INTERVALS
```

DEFMER

Equate DEFMERS to a segment number.

DEFMER

```
MERPAR   DEFxx   s,30        MERGE PARAMETER LIST
MERIFB   DEFxx    MERPAR,1   INPUT FLAG BYTE
MERIF0M  EQUATE X'80'              BIT ON ---> DESCENDING KEYS
MERIF6M  EQUATE X'02'              BIT ON ---> INPUT BLOCK 1 NULL
MERIF7M  EQUATE X'01'              BIT ON ---> INPUT BLOCK 2 NULL
MERRFB   DEFxx   s,1         RETURN FLAG BYTE
MERRF0M  EQUATE X'80'              OUTPUT BLOCK FULL
MERRF1M  EQUATE X'40'              INPUT BLOCK 1 EMPTY
MERRF2M  EQUATE X'20'              INPUT BLOCK 2 EMPTY
MERRF3M  EQUATE X'10'              SEQ CHK NOT ON MERGE UNIT BOUNDARY
MERRF4M  EQUATE X'08'              SEQ CHK ON MERGE UNIT BOUNDARY
MERRF5M  EQUATE X'04'              BOTH INPUT BLOCKS NULL
MERI1B   DEFxx   s,2         INPUT BLOCK 1 BEGIN DISP
MERI1E   DEFxx   s,2         INPUT BLOCK 1 END DISP
MERI2B   DEFxx   s,2         INPUT BLOCK 2 BEGIN DISP
MERI2E   DEFxx   s,2         INPUT BLOCK 2 END DISP
MERDLN   DEFxx   s,2         DATA ITEM LENGTH
MERKYD   DEFxx   s,2         DISP TO KEY IN DATA ITEM
MERKYL   DEFxx   s,1         KEY LENGTH
MERIS1   DEFxx   s,1         INPUT BLOCK 1 SEGMENT NUMBER
MERIS2   DEFxx   s,1         INPUT BLOCK 2 SEGMENT NUMBER
MEROTS   DEFxx   s,1         OUTPUT BLOCK SEGMENT NUMBER
MEROTB   DEFxx   s,2         OUTPUT BLOCK BEGIN DISP
MEROTE   DEFxx   s,2         OUTPUT BLOCK END DISP
MERI1C   DEFxx   s,2         INPUT BLOCK 1 CURRENT DISPLACEMENT
MERI2C   DEFxx   s,2         INPUT BLOCK 2 CURRENT DISPLACEMENT
MEROTC   DEFxx   s,2         OUTPUT BLOCK CURRENT DISPLACEMENT
MERUNT   DEFxx   s,2         MERGE UNIT SIZE
```

DEFREG

DEFREG

```
* * *    REGISTER EQUATES

R00      EQUATE    0
R01      EQUATE    1
R02      EQUATE    2
R03      EQUATE    3
R04      EQUATE    4
R05      EQUATE    5
R06      EQUATE    6
R07      EQUATE    7
R08      EQUATE    8
R09      EQUATE    9
R10      EQUATE   10
R11      EQUATE   11
R12      EQUATE   12
R13      EQUATE   13
R14      EQUATE   14
R15      EQUATE   15
```

DEFRGS

DEFRGS

```
* * *   REGISTER SECTION OF SEGMENT 0

REGS    DEFLD   0,0,96
REG0    DEFLD   0,0,6
REG1    DEFLD   0,,6
REG2    DEFLD   0,,6
REG3    DEFLD   0,,6
REG4    DEFLD   0,,6
REG5    DEFLD   0,,6
REG6    DEFLD   0,,6
REG7    DEFLD   0,,6
REG8    DEFLD   0,,6
REG9    DEFLD   0,,6
REG10   DEFLD   0,,6
REG11   DEFLD   0,,6
REG12   DEFLD   0,,6
REG13   DEFLD   0,,6
REG14   DEFLD   0,,6
REG15   DEFLD   0,,6
```

DEFSCA

Equate DEFSCAS to a segment number.

DEFSCA

```
SCAPAR  DEFxx   s,18        SCALE PARAMETER LIST
SCALEN  DEFxx    SCAPAR,1   LENGTH OF OUTPUT AREA
SCACHR  DEFxx   s,1         SCALE CHARACTER
SCAFAC  DEFxx   s,1         SCALE FACTOR
SCAINP  DEFxx   s,1         INPUT FLAG BYTE
SCAHDR  DEFxx   s,6         SPECIAL HEADER CHARACTERS
SCADEL  DEFxx   s,4         DELETE CHARACTERS
SCARES  DEFxx   s,3         RESERVED (VALUE MUST BE 3X'00')
SCASIG  DEFxx   s,1         NUMBER OF SIGNIFICANT DIGITS
SCABEG  DEFxx   s,0         BEGINNING OF OUTPUT AREA
```

DEFSCP

Equate DEFSCPS to a segment number.

DEFSCP

```
SCPSTR   DEFxx   s,0     START OF PARAMETER LIST

SCPFC    DEFxx   s,1       FUNCTION REQUESTED

SCPINTR  EQUATE X'01'      INITIALIZE SPA REQUEST
SCPADDR  EQUATE X'02'      ADD ELEMENT REQUEST
SCPRPLR  EQUATE X'03'      REPLACE ELEMENT REQUEST
SCPRTRR  EQUATE X'04'      RETRIEVE ELEMENT REQUEST
SCPRTUR  EQUATE X'05'      RETRIEVE ELEMENT FOR UPDATE REQUEST
SCPDLER  EQUATE X'06'      DELETE AN ELEMENT REQUEST
SCPDLAR  EQUATE X'07'      DELETE ALL ELEMENTS REQUEST
SCPARUR  EQUATE X'25'      ADD OR RETRIEVE FOR UPDATE REQUEST
SCPARLR  EQUATE X'23'      ADD OR REPLACE REQUEST
SCPEXCR  EQUATE X'08'      EXCHANGE DATA AREA WITH THE ELEMENT
```

```
SCPTYP   DEFxx   s,1        ADDRESSING TYPE

SCPKEYM  EQUATE X'00'            KEYED ADDRESSING
SCPELMA  EQUATE X'01'            ELEMENT ADDRESSING

SCPSPID  DEFxx   s,1        SPA ID
SCPRC    DEFxx   s,1        SCRATCH PAD RETURN CODE

SCPRC80  EQUATE X'80'
SCPRC40  EQUATE X'40'
SCPRC20  EQUATE X'20'
SCPRC10  EQUATE X'10'
SCPRC08  EQUATE X'08'
SCPRC04  EQUATE X'04'
SCPRC02  EQUATE X'02'
SCPRC01  EQUATE X'01'
SCPRC00  EQUATE X'00'

SCPELMN  DEFxx   s,2        ELEMENT NUMBER ADDRESS
SCPKEYL  DEFxx   s,1        KEY LENGTH
SCPDATPT DEFxx   s,3        DATA AREA ADDRESS (SEGMENT,
                            DISPLACEMENT)

SCPELMLN DEFXX    SCPELMN,2 ELEMENT LENGTH
SCPELMNB DEFxx   s,2        NUMBER OF ELEMENTS ALLOCATED

SCPDSG   DEFXX    SCPDATPT,1
SCPDDSP  DEFxx   s,2
SCPEND   DEFxx   s,0    END OF PARAMETER LIST
SCPPRL   DEFXX    SCPSTR,(D:SCPEND-D:SCPSTR)
```

DEFSEG

DEFSEG

```
* * *    SEGMENT EQUATES

S00      EQUATE    0
S01      EQUATE    1
S02      EQUATE    2
S03      EQUATE    3
S04      EQUATE    4
S05      EQUATE    5
S06      EQUATE    6
S07      EQUATE    7
S08      EQUATE    8
S09      EQUATE    9
S10      EQUATE   10
S11      EQUATE   11
S12      EQUATE   12
S13      EQUATE   13
S14      EQUATE   14
S15      EQUATE   15
```

DEFSKP

Equate DEFSKPS to a segment number.

DEFSKP

```
SKPPAR     DEFxx    s,18              LSEEKP PARAMETER LIST
SKPFLG1    DEFxx     SKPPAR,1         LSEEKP OPTION FLAG
SKPFPA     EQUATE X'80'                     TABLE/PARM LIST MATCH
                                              ADDRESS
*                                            = 0 USE TABLE MATCH
                                              ADDRESS
*                                            = 1 USE PARM LIST MATCH
                                              ADR
SKPFBS     EQUATE X'40'                     SEQUENT/BINARY TABLE
                                              SEARCH
*                                            = 0 SEQUENTIAL TABLE
                                              SEARCH
*                                            = 1 BINARY TABLE SEARCH
SKPFTI     EQUATE X'20'                     TABLE LOCATION
*                                            = 0 IN NON-SPLIT AP OR
*                                                CONSTANT PORTION OF
                                                 SPLIT
*                                            = 1 IN INSTRUCTION POR-
*                                                TION OF A SPLIT AP
SKPFCE     EQUATE X'10'                     NO COPY/COPY TABLE ELE-
                                               MENT
*                                            = 0 DO NOT COPY TABLE
                                               ELEMENT
*                                            = 1 COPY TABLE ELEMENT
SKPFEQ     EQUATE X'08'                     LOCATE EQUAL TABLE ENTRY
*                                            (REQUIRED SETTING = 1)
*                                            = 1 RETURN EQUAL TABLE
                                              ENTRY
SKPFBL     EQUATE X'04'                     BINARY RETURN < IF NOT
                                              EQUAL
*                                            = 0 DO NOT RETURN LESS
                                              THAN
*                                            = 1 RETURN IF < IF NOT
                                              EQUAL
SKPFBG     EQUATE X'02'                     BINARY RETURN > IF NOT
                                              EQUAL
*                                            = 0 DO NOT RETURN GRTR
                                              THAN
                                             = 1 RETURN GRTR THAN
                                              IF NO =

SKPFLG2    DEFxx    s,1               LSEEKP OPTION FLAG BYTE 2
SKPFNS     EQUATE X'80'                     DO NOT/ DO RETURN NSI
*                                            = 0 DO NOT RETURN NSI
*                                            = 1 RETURN NSI
SKPREG     DEFxx    s,1               RETURN NSI IN STACK/REGISTER
*                                            = 0 RETURN NSI IN STACK
*                                            > 0 REGISTER TO CONTAIN
                                                 NSI
SKPARS     DEFxx    s,1               SEGMENT CONTAINING ARGUMENT
SKPARD     DEFxx    s,2               DISPLACEMENT TO ARGUMENT
SKPARL     DEFxx    s,2               LENGTH OF ARGUMENT
SKPSCS     DEFxx    s,1               TABLE SEGMENT
SKPSCD     DEFxx    s,2               DISPLACEMENT TO TABLE
SKPMTH     DEFxx    s,2               MATCH ADDRESS
SKPENS     DEFxx    s,1               SEGMENT CONTAINING COPY FIELD
SKPEND     DEFxx    s,2               DISPLACEMENT TO COPY FIELD
SKPENL     DEFxx    s,2               LENGTH OF COPY FIELD
```

DEFSMS

```
**********************************************************************
*                                                                    *
*        SEGMENT ONE MACHINE SECTION DEFINITIONS                     *
*                                                                    *
*          GENERAL FIELDS AND EQUATES SUBSECTION:                    *
*                                                                    *
**********************************************************************
         LSPACE
SMSPSW   DEFLD     1,0,4       PSW
SMSSID   DEFLD     1,0,1       STATION ID
SMSCCD   DEFLD     1,,1        COMPLETION CODE
SMSUIC   DEFLD     1,,2        RELATIVE INSTRUCTION COUNTER
SMSPCA   DEFLD     1,,2        ADDRESS OF INSTRUCTION THAT CAUSED PC
SMSPCC   DEFLD     1,,1        PROGRAM CHECK CODE
SMSABK   DEFLD     1,,1        OPB A/B INDICATION
SMSIML   DEFLD     1,,2        I/P MESSAGE LENGTH
SMSICT   DEFLD     1,SMSIML,2  COUNT OF IMPLIED DATA SETS OPENED
SMSDST   DEFLD     1,,2        DEVICE STATUS BYTES
SMSDS1   DEFLD     1,SMSDST,1  STATUS FIELD 1
SMSDS2   DEFLD     1,,1        STATUS FIELD 2
SMSLTC   DEFLD     1,,2        LOOP THRESHOLD COUNT
SMSLTH   DEFLD     1,,2        LOOP THRESHOLD VALUE
         LSPACE
SMSIND   DEFLD     1,,1        INDICATOR BYTE:
SMSDATSM EQUATE    X'10'         STATION CONFIGURED FOR DATSM USE MASK
SMSICAM  EQUATE    X'40'         CANCEL MASK
SMSICAO  EQUATE    X'BF'         TURN OFF CANCEL FLAG MASK
         LSPACE
SMSDCB   DEFLD     1,,1        DELIMITER CONTROL BYTE
SMSDEL   DEFLD     1,,2        ALTERNATE DELIMITER TABLE ADDR
SMSDSS   DEFLD     1,,1        SWITCHED MSG STB ID FIELD
         LSPACE
SMSAFL   DEFLD     1,,1        ASYNCHRONOUS FLAG FIELD:
SMSACP   DEFLD     1,SMSAFL,1    ASYNCHRONOUS CPU INTERRUPT FLAG
SMSACPM  EQUATE    X'80'         ASYNCHRONOUS CPU INTERRUPT MASK
SMSAST   DEFLD     1,SMSAFL,1    ASYNCHRONOUS STATION INTERRUPT FLAG
SMSASTM  EQUATE    X'40'         ASYNCHRONOUS STATION INTERRUPT MASK
SMSAAP   DEFLD     1,SMSAFL,1    ASYNCHRONOUS ALA INTERRUPT FLAG
SMSAAPM  EQUATE    X'20'         ASYNCHRONOUS ALA INTERRUPT MASK
         LSPACE
SMSTMR   DEFLD     1,,4        SMS TIMER FIELD
SMSHTM   DEFLD     1,SMSTMR,2  HIGH ORDER 2 BYTES OF TIMER
SMSLTM   DEFLD     1,,2        LOW ORDER 2 BYTES OF TIMER
SMSPCT   DEFLD     1,,2        PAUSE INSTRUCTION COUNTER
         LSPACE
SMSFG2   DEFLD     1,,1        FLAG BYTE:
SMSPCRM  EQUATE    X'80'         PROGRAM CHECK ROUTINE IN CONTROL
SMSLWSM  EQUATE    X'40'         LOGICAL WAIT STATE
         LSPACE
SMSLSB   DEFLD     1,,1        PARENT AP LINK STACK USE COUNT
SMSWAIT  DEFLD     1,,1        WAIT INSTRUCTION TERMINATING CONDITION
SMSICPM  EQUATE    X'10'         CPU MESSAGE
SMSIAPM  EQUATE    X'20'         ALA MESSAGE
SMSITPM  EQUATE    X'30'         TERMINAL MESSAGE
SMSISPM  EQUATE    X'40'         SWITCHED MESSAGE
SMSIPPM  EQUATE    X'50'         PROGRAM INERRUPT VIA POST INSTRUCTION
SMSITFM  EQUATE    X'60'         TIMER INTERRUPT
SMSIATM  EQUATE    X'70'         ATTENTION INTERRUPT
         LSPACE
SMSDRG   DEFLD     1,,1        REGISTER FOR DELIMITER TABLE ADDRESS
         DEFLD     1,,2        RESERVED
         LSPACE
```

```
*************************************************************
*                                                           *
*          TERMINAL FIELDS AND EQUATES SUBSECTION:          *
*                                                           *
*                                                           *
*************************************************************
         LSPACE
SMSEID   DEFLD     1,,1       EOM IDENTIFIER
SMSECT   DEFLD     1,,1       NUMBER OF CHARS IN THE CURRENT EOM
                                STRING
SMSTGU   DEFLD     1,,1       ADDRESS OF TERMINAL GROUP UNIT
SMSSSU   DEFLD     1,SMSTGU,1 ADDRESS SHARED SLOT UNIT (SUB-ADDR)
         LSPACE
SMSSAM   DEFLD     1,,1       LDA ATTENTION SUMMARY MASK:
SMSLDA0  EQUATE    X'80'        MESSAGE AVAILABLE ON LDA-0
SMSLDA1  EQUATE    X'40'        MESSAGE AVAILABLE ON LDA-1
SMSLDA2  EQUATE    X'20'        MESSAGE AVAILABLE ON LDA-2
SMSLDA3  EQUATE    X'10'        MESSAGE AVAILABLE ON LDA-3
SMSLDA4  EQUATE    X'08'        MESSAGE AVAILABLE ON LDA-4
SMSLDA5  EQUATE    X'04'        MESSAGE AVAILABLE ON LDA-5
SMSLDA6  EQUATE    X'02'        MESSAGE AVAILABLE ON LDA-6
SMSLDA7  EQUATE    X'01'        MESSAGE AVAILABLE ON LDA-7
         LSPACE
SMSSPR   DEFLD     1,,2       RESERVED
SMSCUR   DEFLD     1,SMSSPR,2 LOCAL KEYTRACKING CURSOR ADDRESS
SMSMSL   DEFLD     1,,1       LENGTH OF MAGNETIC STRIPE DATA
SMSKSM   DEFLD     1,,1       LDA FIRST KEYSTROKE SUMMARY MASK:
         LSPACE
```

```
******************************************************************** *
*                                                                    *
*          DISK/DISKETTE FIELD AND EQUATES SUBSECTION:               *
*                                                                    *
*                                                                    *
******************************************************************** *
         LSPACE
SMSUNK   DEFLD     1,,2          UNIQUE ID FOR UNKEYED DATA SET
SMSFG1   DEFLD     1,,1          FLAG BYTE:
SMSMBTM  EQUATE    X'40'           MULTI-BLOCK I/O
SMSNXRM  EQUATE    X'20'           AVOID RECORD TRANSFER
SMSDSKM  EQUATE    X'01'           FLAG: SET-->DISK,  RESET-->DISKETTE
SMSDT2M  EQUATE    X'02'           ALTERNATE DRIVE - DISK OR DISKETTE
SMSDKTI  EQUATE    X'0F'           MASK TO ISOLATE DRIVE SELECT BITS
         LSPACE
SMSDKT1  EQUATE    X'00'           SELECT BIT VALUE FOR DISKETTE DRIVE 1
SMSDKT2  EQUATE    X'02'           SELECT BIT VALUE FOR DISKETTE DRIVE 2
         LSPACE
SMSDSK1  EQUATE    X'01'           SELECT BIT VALUE FOR DISK DRIVE A
SMSDSK2  EQUATE    X'03'           SELECT BIT VALUE FOR DISK DRIVE B
SMSDSK3  EQUATE    X'05'           SELECT BIT VALUE FOR DISK DRIVE C
SMSDSK4  EQUATE    X'07'           SELECT BIT VALUE FOR DISK DRIVE D
SMSDID   DEFLD     1,,1          DATA SET ID
         LSPACE
SMSRPS   DEFLD     1,,4          RECORD SEQ NUMBER FOR READ/REPLACE
SMSRSNH  DEFLD     1,SMSRPS,2      1ST HWD OF RECORD SEQ NUMBER
SMSRSNL  DEFLD     1,,2            2ND HWD OF RECORD SEQ NUMBER
SMSRSN   DEFLD     1,SMSRSNL,2     2ND HWD OF RECORD SEQ NUMBER
SMSRSN1  DEFLD     1,SMSRSN,1      3RD BYTE OF RECORD SEQ NUMBER
SMSRSN2  DEFLD     1,,1            4TH BYTE OF RECORD SEQ NUMBER
         LSPACE
SMSSFW   DEFLD     1,,1          SUB-FILE ID FOR TEMP FILE WRITE
SMSSFR   DEFLD     1,,1          SUB-FILE ID FOR TEMP FILE READ/REPLACE
SMSKEY   DEFLD     1,,4          CURRENT POSITION OF KEYED DATA SET
SMSKEY1  DEFLD     1,SMSKEY,2    FIRST  HWD OF 4 BYTE KEY
SMSKEY2  DEFLD     1,,2          SECOND HWD OF 4 BYTE KEY
         DEFLD     1,,2          RESERVED
SMSFSN   DEFLD     1,,2          FILE SEQ NUM FROM LAST TEMP FILE WRITE
         DEFLD     1,,2          RESERVED
SMSSSN   DEFLD     1,,2          SUB-FILE SEQ NUM FROM LAST TEMP FILE WRT
         DEFLD     1,,2          RESERVED
SMSCSN   DEFLD     1,,2          COMPOSITE SEQ NUM FROM LAST TEMP FILE WRT
         LSPACE
SMSNDB   DEFLD     1,,4          NUM OF D.S. BLKS AVAIL FOR LWRITE
SMSADS1  DEFLD     1,SMSNDB,2      1ST HWD OF D.S. BLKS AVAIL FOR LWRITE
SMSADS   DEFLD     1,,2            2ND HWD OF D.S. BLKS AVAIL FOR LWRITE
         LSPACE
```

```
*******************************************************
*                                                     *
*         CPU FIELDS AND EQUATES SUBSECTION:          *
*                                                     *
*******************************************************
         LSPACE
SMSCRC   DEFLD      1,,2        CPU READ CONTROL FIELDS
SMSCRF   DEFLD      1,SMSCRC,1  CPU READ FLAGS
SMSBRL   DEFLD      1,SMSCRC,1  BSC READ CONTROL
SMSCRT   DEFLD      1,,1        CPU READ TYPE
SMSCST   DEFLD      1,,1        LINK STATUS BYTE
SMSCCR   EQUATE     X'04'       OPERATIONAL CIRCUIT FLAG
SMSCID   DEFLD      1,SMSCST,1  S/S UNIT ID
SMSTR2   DEFLD      1,,1        RESERVED
SMSCWC   DEFLD      1,,2        CPU WRITE CONTROL FIELDS
SMSCWF   DEFLD      1,SMSCWC,1  CPU WRITE FLAGS
SMSBWC   DEFLD      1,SMSCWC,1  BSC WRITE CONTROL
SMSCWT   DEFLD      1,,1        CPU WRITE TYPE
SMSCRS   DEFLD      1,,2        READ SEQ NUMBER OR ID
SMSBIH   DEFLD      1,SMSCRS,2  INPUT HEADER LENGTH
SMSCWS   DEFLD      1,,2        WRITE SEQ NUMBER OR ID
SMSBOH   DEFLD      1,SMSCWS,2  OUTPUT HEADER LENGTH
SMSCSR   DEFLD      1,,2        READ RESPONSE TO DATA SEQ NUMBER
SMSCHL   DEFLD      1,SMSCSR,2  HEADER LENGTH
SMSCPE   DEFLD      1,,2        CPU READ/WRITE FLAGS EXTENSION
SMSCRE   DEFLD      1,SMSCPE,1  CPU READ FLAGS EXTENSION
SMSCWE   DEFLD      1,,1        CPU WRITE FLAGS EXTENSION
         DEFLD      1,,4        RESERVED
         DEFLD      1,,1        RESERVED
SMSAMS   DEFLD      1,,3        ALA MACHINE SEGMENT, SEG NO./DIS-
                                  PLACEMENT
SMSAMS1  DEFLD      1,SMSAMS,1  AMS SEGMENT NUMBER
SMSAMS2  DEFLD      1,,2        AMS SEGMENT DISPLACEMENT
         LSPACE

******************************************************
*                                                    *
*         LINKSTACK FIELDS SUBSECTION:               *
*                                                    *
*                                                    *
******************************************************
         LSPACE
SMSLSH   DEFLD      1,,2        LINK STACK HEADER
SMSLSM   DEFLD      1,SMSLSH,1  MAXIMUM NUMBER OF ENTRIES
SMSLSE   DEFLD      1,,1        CURRENT NUMBER OF ENTRIES
*                               SEE SMSLSB FOR BASE COUNT OF STACK
SMSLSA   DEFLD      1,,12       LINK STACK AREA
```

Segment 1 Fields

Segment 1 contains the following fixed fields:

Station ID (SMSSID): The number (2 through 60) of the logical work station. This field is regenerated each time an LEXIT instruction is issued.

Condition Code (SMSCCD): A code indicating the result of execution of an instruction (bits 4-7 of the byte at SMSCCD). Not all instructions set a condition code.

Instruction Counter (SMSUIC): The displacement, into Segment 14, of the next instruction to be executed. This field is set to 0 when an LEXIT instruction is executed. The instruction counter may be changed by the application program to alter program flow, although a branch or LSEEK instruction is preferable.

Program Check Address (SMSPCA): The displacement, into Segment 14, of the instruction that caused a program check. This field is only set when a program check occurs.

Program Check Code (SMSPCC): A code indicating the cause of the program check. This field is set before control is given to the application program's program-check routine.

Segment 0 Selected (SMSABK): A character A or B (hex C1 or hex C2) that indicates which Segment 0 is desired when a SELECT instruction is executed.

Input Message Length (SMSIML): The length of data placed in the station's segment as a result of an LREAD instruction or the amount of data *not* transmitted (residual length) by an LWRITE, WRTI, or REPLACE instruction.

Implied Data Set Opens (SMSICT): A count of data sets opened as a result of opening the associated keyed data set.

Device Status (SMSDST): A code indicating the status that resulted from data transmission or the reason a data transmission operation failed.

Instruction Count (SMSLTC): The number of instructions executed by the application program since the last LEXIT instruction was executed.

Instruction Count Threshold (SMSLTH): The maximum number of instructions to be executed before the controller assumes that the station's application program is in an unending loop. When the instruction count (SMSLTC) exceeds the value in this field, the controller causes a program check to occur for this station.

Indicator Byte (SMSIND): Bit 1 of this byte is set to 1 when the reset key on a terminal assigned to LDA 0 is pressed twice (an attention indication). Other features set this byte to different values. This byte is reset to 0 when an LEXIT instruction is executed.

Delimiter Control Byte (SMSDCB): A mask used to select the set of delimiters to be recognized by the controller.

Alternate Delimiter Table Address (SMSDEL): Address of the delimiter table currently in use when multiple delimiter tables are defined for the application program.

Switched Message Station ID (SMSDSS): For a station-to-station write or check of a station-to-station write, the number of the station to receive the data. For a station-to-station read, the number of the station that sent the data.

Asynchronous Flag Field (SMSAFL): A one-byte field whose two high-order bits refer to station and CPU interruptions pending. By scanning this field before relinquishing control, a work station can determine if any messages are pending for it.

Station Timer Value (SMSTMR): The value (in seconds) that is compared with the controller timer value (GMSTMR) to determine if the work station should be dispatched at its asynchronous timer entry point. If zero, the station is not dispatched.

Pause Instruction Counter (SMSPCT): A count of the number of times PAUSE has been performed since the last LEXIT.

Flag Field (SMSFG2): A 1-byte flag field indicating the following:

Bit Position	Meaning
1xxx xxxx	A program check routine is currently being executed.
x1xx xxxx	The station is currently in a logical wait state.

Link Stack Base Entry (SMSLSB): The number of the last link stack entry used by the parent application program.

Wait Terminating Condition (SMSWAIT): A 1-byte field containing the condition ending the LWAIT instruction. If more than one ending condition is present, only the highest priority condition is indicated.

Delimiter Table Address Register (SMSDRG): The number of the application program register that contains the address of the delimiter table.

End-of-Message/End-of-File ID (SMSEID): The translate value defined for a key designated as an EOM/EOF key with the EID option.

Number of Characters in EOM (SMSECT): The number of characters placed in the input segment as a result of translating the EOM key (0-6). SMSIML minus SMSECT is the length of the data entered.

Terminal Group Unit Address (SMSTGU): The subaddress (hex 00 through hex FF) identifying the individual unit currently in use in a terminal group.

LDA Attention Summary (SMSSAM): A one-byte field whose bits correspond to logical device addresses (the high-order bit corresponds to LDA0; the low-order bit, to LDA7). A bit set to 1 indicates a message pending from the device associated with that LDA.

Magnetic Stripe Reader Data Length (SMSMSL): The length of the data string received from the magnetic stripe reader.

Unkeyed Data Set ID (SMSUNK): A 2-byte field containing a unique ID of an unkeyed data set.

Disk/Diskette Flag Field (SMSFG1): A 1-byte field containing flags that the application program sets to control disk and diskette operations. This field contains a flag that controls single and multi-block operations; a flag that controls data transfer for keyed operations; and flags that select the disk or diskette drive to be used.

Disk Data Set ID (SMSDID): The data set ID (DSID) of the disk or diskette data set to be accessed by the controller application program. This field is set by the application program prior to issuing an instruction to the disk or diskette (unless previously set to the desired ID).

Record Sequence Number (SMSRPS): A 4-byte field describing the data record position for disk and diskette operations. The low-order 2 bytes are named SMSRSN for compatibility with 3600.

Subfile Index for Disk/Diskette Write (SMSSFW): The subfile number during a write to the temporary file.

Subfile Index for Disk/Diskette Read (SMSSFR): The subfile number during read from the temporary file.

Keyed Data Set Position (SMSKEY): A 4-byte field indicating the current position in a keyed data set.

File Sequence Number from Last Diskette/Disk Write (SMSFSN): The file sequence number from the last write to the temporary file or the log.

Subfile Sequence Number from Last Diskette/Disk Write (SMSSSN): The subfile sequence number from the last write to the temporary file.

Composite File Sequence Number from Last Diskette/Disk Write (SMSCSN): The composite file sequence number from the last write to the temporary file.

Available Data Set Blocks (SMSADS1/ADS): A 4-byte field indicating the number of data set blocks available for use by LWRITE.

Host Processor Read Control Field (SMSCRF, SMSCRT): The type of message received from the central processor and any flags that accompanied the message.

Link Status (SMSCST): The status of the station's use of the communication link.

Host Write Control Field (SMSCWF, SMSCWT): The type of message being sent by the station and any flags that accompany the message.

Host Read Sequence Number (SMSCRS): The sequence number of the last message received from the host.

Host Write Sequence Number (SMSCWS): The sequence number of the last message sent to the host.

Host Read-Response Sequence Number (SMSCSR): The sequence number of the message being responded to.

CPU Read/Write Extension (SMSCPE): A 2-byte flag field used as extensions when performing CPU LREAD or LWRITE. SMSCRE is used as an extension of SMSCRF for LREAD and SMSCWE is an extension of SMSCWF for LWRITE.

ALA Machine Segment Address (SMSAMS): A 3-byte field containing the address of the ALA Machine Segment. The first byte indicates the segment, and the remaining 2 bytes are the displacement of the AMS.

Link Stack Maximum (SMSLSM): The maximum number of entries that the link stack can contain. Initialized by the RETSTK operand of the STATION macro.

Number of Return Stack Entries (SMSLSE): The number of current entries in the return address stack. The number is incremented each time a BRANL or BRANLR adds an entry to the stack and decremented each time an LRETURN uses an entry.

Return Address Stack (SMSLSA): Beginning of the area in Segment 1 where return addresses will be stored. The number of bytes reserved is two times the number of return addresses indicated in SMSLSM. Addresses are added to the stack in the next available field when the appropriate type of branch-and-link instruction is executed.

DEFSOR

Equate DEFSORS to a segment number.

DEFSOR

```
SORPAR   DEFxx    s,17         SORT PARAMETER LIST
SORIFB   DEFxx     SORPAR,1    INPUT FLAG BYTE
SORIFOM  EQUATE X'80'                BIT ON ---> DESCENDING KEYS
SORRFB   DEFxx    s,1          RESERVED
SORSBG   DEFxx    s,2          SORT BLOCK BEGIN
SORSND   DEFxx    s,2          SORT BLOCK END
SORWBG   DEFxx    s,2          WORK BLOCK BEGIN
SORWND   DEFxx    s,2          WORK BLOCK END
SORDLN   DEFxx    s,2          DATA ITEM LENGTH
SORKYD   DEFxx    s,2          DISPLACEMENT TO KEY IN DATA ITEM
SORKYL   DEFxx    s,1          KEY LENGTH
SORSSG   DEFxx    s,1          SEGMENT NUMBER CONTAINING SORT BLOCK
SORWSG   DEFxx    s,1          SEGMENT NUMBER CONTAINING WORK BLOCK
```

DEFTRP

Equate DEFTRPS to a segment number.

DEFTRP

```
TRPPAR   DEFxx    s,30         PARM LIST (INCLUDES 'TRPENT1')
TRPINS   DEFxx     TRPPAR,2    INPUT AREA'S SEGMENT
TRPIND   DEFxx    s,2          INPUT AREA'S DISPLACEMENT
TRPINL   DEFxx    s,2          INPUT AREA'S LENGTH
TRPOUS   DEFxx    s,2          OUTPUT AREA'S SEGMENT
TRPOUD   DEFxx    s,2          OUTPUT AREA'S DISPLACEMENT
TRPOUL   DEFxx    s,2          OUTPUT AREA'S LENGTH
TRPMSK   DEFxx    s,1          BREAK CONTROL MASK
TRPNTT   DEFxx    s,1          NUMBER OF TRANSLATE TABLES
TRPPTT   DEFxx    s,1          PRIOR TRANSLATE TABLE NUMBER
TRPCTT   DEFxx    s,1          CURRENT TRANSLATE TABLE NUMBER
TRPLIC   DEFxx    s,1          LAST INPUT CODE PROCESSED
TRPFNC   DEFxx    s,1          FUNCTION CODE
TRPLID   DEFxx    s,2          LAST INPUT CODE DISPLACEMENT
TRPLOD   DEFxx    s,2          LAST INPUT CODE'S OUTPUT DISPLACEMENT
TRPCNT   DEFxx    s,2          TOTAL NUMBER OF CHARACTERS OUTPUT
TRPTST   DEFxx    s,1          TERMINATION STATUS FIELDS
TRPTBR   EQUATE X'80'                  TRANSLATE BREAK
TRPTOO   EQUATE X'40'                  OUTPUT OVERFLOW
TRPTIO   EQUATE X'20'                  INPUT OVERFLOW
TRPTII   EQUATE X'10'                  INVALID INPUT CODE
TRPTNL   EQUATE X'08'                  INITIAL OUTPUT LENGTH WAS ZERO
         DEFxx    s,1          RESERVED
TRPTTL   DEFxx    s,0          TRANSLATE TABLE LIST
TRPENT1  DEFxx    s,4          ENTRY1 OF XLATE TABLE LIST
TRPTTS1  DEFxx     TRPENT1,2     XLATE TABLE 1'S SEGMENT
TRPTTD1  DEFxx    s,2            XLATE TABLE 1'S DISPLACEMENT
```

DEFTRT

Equate DEFTRTS to a segment number.

DEFTRT

```
TRTHDR   DEFxx   s,12        TRANSLATE TABLE HEADER
TRTLEN   DEFxx    TRTHDR,2   LENGTH OF WHOLE TABLE
TRTDOP   DEFxx   s,2         DISP: TBL-HDR TO OFF POINTED STRINGS
TRTLOW   DEFxx   s,1         LOW CODE IN TABLE
TRTELN   DEFxx   s,1         LENGTH OF ENTRY
TRTHFG   DEFxx   s,1         HEADER FLAGS
TRTTRO   EQUATE X'80'                ONLY TRANSLATE ENTRIES
TRTMDS   EQUATE X'40'                MODE = SKIP (FOR OUT OF RANGE)
TRTMDM   EQUATE X'20'                MODE = MOVE (FOR OUT OF RANGE)
TRTMDU   EQUATE X'10'                MODE = USE (FOR OUT OF RANGE)
TRTUSE   DEFxx   s,1        VALUE FOR 'USER' MODE
TRTNUM   DEFxx   s,2        NUMBER OF CODES IN THE TABLE
         DEFxx   s,2        RESERVED

*             TRANSLATE TABLE ENTRIES' FLAG BYTE

TRTEFG   DEFxx   s,0         ENTRIES' FLAG BYTE:
TRTOPS   EQUATE X'80'                VALUES IN OFF-POINTED STRING
TRTOCS   EQUATE X'40'                OUTPUT CHARACTER STRING
                                       DEFINED
TRTFNC   EQUATE X'20'                FUNCTION DEFINED
TRTBRK   EQUATE X'10'                BREAK DEFINED
TRTBAD   EQUATE X'08'                BREAK ADDRESS DEFINED
TRTINL   EQUATE X'04'                INLEN DEFINED TO BE OTHER THAN

TRTRES1  EQUATE X'02'                RESERVED  (MUST BE ZERO)
TRTTOC   EQUATE X'01'                2 OUTPUT CHAR'S IN TABLE ENTRY
```

DEFTSX

Equate DEFTSXS to a segment number.

DEFTSX

```
TSXPAR   DEFxx   s,5          'TESTX' PARAMETER LIST
TSXFLG   DEFxx    TSXPAR,1    FLAG BYTE:
TSXFREM  EQUATE    X'80'                REQUEST SEGMENT'S STATUS
TSXFSTM  EQUATE    X'02'                STATION ACTIVE FOR
                                          INDEXING
TSXFSEM  EQUATE    X'01'                SEGMENT ACTIVE FOR
                                          INDEXING
TSXSID   DEFxx   s,1          STATION ID
TSXSEG   DEFxx   s,1          SEGMENT NUMBER
TSXADR   DEFxx   s,2          INDEX REG-NUM TBL'S ADDRESS
TSXREG1  DEFxx    TSXADR,1    OPERAND ONE'S INDEX REGISTER
TSXREG2  DEFxx   s,1          OPERAND TWO'S INDEX REGISTER
```

DEFVUE

Equate DEFVUES to a segment number.

DEFVUE

```
VUEPAR   DEFxx    s,26          VIEW PARAMETER LIST
VUEREQ   DEFxx     VUEPAR,1     REQUEST CODE
VUESTA   DEFxx    s,1           STATION NUMBER
VUESTK   DEFxx    s,1           STACK ID
VUEFG1   DEFxx    s,1           FLAG BYTE 1
VUEFL0M  EQUATE     X'80'                  STACK ENTRY IN USE
VUEFL1M  EQUATE     X'40'                  STACK ENTRY PERMANENT
VUEFL2M  EQUATE     X'20'                  'B' SET OF REGISTERS ACTIVE
VUEUIC   DEFxx    s,2           USER INSTRUCTION COUNTER
VUELSB   DEFxx    s,1           BOTTOM OF SMS LINK STACK
VUELSE   DEFxx    s,1           TOP OF SMS LINK STACK
VUEDEL   DEFxx    s,2           SAVED VALUE OF SMSDEL
VUEPNT   DEFxx    s,1           PARENT SEGMENT SPACE ID
VUEFG2   DEFxx    s,1           FLAG BYTE 2
VUERTF   EQUATE     X'80'                  AP TRANSIENT FLAG
VUEFAC   EQUATE     X'01'                  AP CALLABLE
VUEPID   DEFxx    s,8           APPLICATION PROGRAM NAME
VUERES1  DEFxx    s,6           RESERVED
```

Appendix C. Assembler Error Messages

The following lists the error messages that may result from a 4700 assembly.

BDK900I SYMBOL NOT EQUATED

Explanation: Self-explanatory.

User Response: Equate the symbol to a value or delete the symbol; reassemble.

BDK901I LABEL MISSING

Explanation: Self-explanatory.

User Response: Add the missing label and reassemble.

BDK902I INCORRECT NUMBER OF OPERANDS

Explanation: The number of operands specified is not correct.

User Response: Check the syntax of the instruction, correct the error, and reassemble.

BDK903I MISSING OPERANDS

Explanation: Required operand or operands have been omitted.

User Response: Check the syntax of the instruction, correct the error, and reassemble.

BDK904I MISSING FIRST OPERAND

Explanation: Self-explanatory.

User Response: Add the first operand to the instruction and reassemble.

BDK905I MISSING SECOND OPERAND

Explanation: Self-explanatory.

User Response: Add the second operand to the instruction and reassemble.

BDK906I MISSING THIRD OPERAND

Explanation: Self-explanatory.

User Response: Add the third operand to the instruction and reassemble.

BDK907I INVALID OPERAND

Explanation: There is a syntax error in one or more operands.

User Response: Check the syntax of the instruction, correct the error, and reassemble.

BDK908I INVALID FIRST OPERAND

Explanation: Self-explanatory.

User Response: Correct the first operand and reassemble.

BDK909I INVALID SECOND OPERAND

Explanation: Self-explanatory.

User Response: Correct the second operand and reassemble.

BDK910I INVALID THIRD OPERAND

Explanation: Self-explanatory.

User Response: Correct the third operand and reassemble.

BDK911I INVALID FOURTH OPERAND

Explanation: Self-explanatory.

User Response: Correct the fourth operand and reassemble.

BDK912I SEGMENT NOT PROPERLY SPECIFIED

Explanation: The segment specification is either nonnumeric or not in the range of 0 to 15, or an undefined symbol was encountered.

User Response: Correct the segment specification and reassemble.

BDK913I REGISTER NOT PROPERLY SPECIFIED

Explanation: The register specification is either nonnumeric or not in the range of 0 to 15.

User Response: Correct the register specification and reassemble.

BDK914I LENGTH NOT PROPERLY SPECIFIED

Explanation: The length specification is either nonnumeric or not within the range valid for the instruction.

User Response: Correct the length specification and reassemble.

BDK915I DISPLACEMENT NOT PROPERLY SPECIFIED

Explanation: The displacement specification is either nonnumeric or not within the range valid for the instruction.

User Response: Correct the displacement specification and reassemble.

BDK916I SEGMENT FOURTEEN CANNOT BE USED

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK918I DATE OPERAND MISSING OR INCORRECT

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK919I MULTIPLE USE OF BEGIN, FINISH, DEFCODE, DEFLINK, OR DEFASEP NOT PERMITTED

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK920I TABLE ENTRY [n] IS INVALID

Explanation: There is a syntax error in positional operand (n) of the TABLE instruction.

User Response: Correct the error and reassemble.

BDK921I MISSING FOURTH OPERAND

Explanation: Self-explanatory.

User Response: Add the fourth operand and reassemble.

BDK922I LABEL OR ADDRESS GREATER THAN 8 CHARACTERS

Explanation: Self-explanatory.

User Response: Correct the label or address and reassemble.

BDK923I GLOBAL LOCATION COUNTER OVERFLOW

Explanation: Application program section is greater than 64K.

User Response: Correct the error and reassemble.

BDK924I LENGTH OPERAND MISSING

Explanation: The LNG keyword has been omitted on a TABLE instruction.

User Response: Correct the error and reassemble.

BDK925I LENGTH TOO LONG

Explanation: The LNG operand on the TABLE instruction is not within the valid range.

User Response: Correct the error and reassemble.

BDK926I CONFLICT: NOT ALL ENTRIES HAVE MATCH ADDRESS

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK927I DUPLICATE LABEL

Explanation: The same label appears on a previous EQUATE, DEFLD, or DEFCON instruction.

User Response: Correct the error and reassemble.

BDK928I OVERLAY MACRO NOT PERMITTED

Explanation: An overlay instruction may occur only after a FINISH or ENDOVLY instruction.

User Response: Correct the error and reassemble.

BDK929I ORIGIN OPERAND OMITTED

Explanation: The ORIGIN operand on an OVLYSEC instruction has been omitted.

User Response: Add the missing operand and reassemble.

BDK930I ORIGIN OPERAND INVALID

Explanation: The ORIGIN operand on an OVLYSEC instruction is invalid.

User Response: Correct the error and reassemble.

BDK931I GLOBAL TABLE OVERFLOW

Explanation: More than 4096 EQUATE, DEFCON, and DEFLD instructions have been coded.

User Response: If using OS/VS the user may request the customer engineer to increase the global tables.

BDK932I LLOAD NOT PERMITTED

Explanation: The LLOAD instruction is not permitted unless the NUMOVLY operand is coded on the BEGIN instruction.

User Response: Correct the error and reassemble.

BDK933I TOO MANY ENTRIES FOR TABLE

Explanation: The number of table entries exceeds the limit allowed.

User Response: Correct the error and reassemble.

BDK934I DEFLINK MACRO NOT SPECIFIED

Explanation: Self-explanatory.

User Response: Code the DEFLINK macro and reassemble.

BDK935I TYPE KEYWORD OPERAND IS INVALID

Explanation: The TYPE operand for the OPENSESS or CLOSSESS macro was specified incorrectly.

User Response: Correct the error and reassemble.

BDK936I SESSION ID OPERAND NOT SPECIFIED

Explanation: The required operand has been omitted, was specified incorrectly, or has never been defined.

User Response: Correct the OPENSESS macro and reassemble.

BDK937I RESP KEYWORD OPERAND IS INVALID

Explanation: The RESP operand was not specified correctly.

User Response: Correct the error and reassemble.

BDK938I BRCKT KEYWORD OPERAND IS INVALID

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK939I DATA KEYWORD OPERAND IS INVALID

Explanation: The DATA operand of the LSEND macro was not specified correctly.

User Response: Correct the error and reassemble.

BDK940I LOSS OF CONTACT ENTRY OPERAND OMITTED

Explanation: The required operand has been omitted, was specified incorrectly, or has never been defined.

User Response: Correct the DEFCODE macro and reassemble.

BDK941I CANCEL KEYWORD OPERAND IS INVALID

Explanation: The CANCEL operand of the LRECEIVE macro was not specified correctly.

User Response: Correct the error and reassemble.

BDK942I OUTPUT SEGMENT OPERAND OMITTED

Explanation: The required operand has been omitted, was specified incorrectly, or has never been defined.

User Response: Correct the error and reassemble.

BDK943I INPUT SEGMENT OPERAND OMITTED

Explanation: The required operand has been omitted, was specified incorrectly, or has never been defined.

User Response: Correct the error and reassemble.

BDK944I WORK SEGMENT OPERAND OMITTED

Explanation: The required operand has been omitted, was specified incorrectly, or has never been defined.

User Response: Correct the error and reassemble.

BDK945I LINK REGISTER OPERAND OMITTED

Explanation: The required operand has been omitted, was specified incorrectly, or has never been defined.

User Response: Correct the error and reassemble.

BDK946I WORK REGISTER OPERAND OMITTED

Explanation: The required operand has been omitted, was specified incorrectly, or has never been defined.

User Response: Correct the error and reassemble.

BDK947I LINK REGISTER SAME AS WORK REGISTER

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK948I LINK REGISTER SAME AS TRACE REGISTER

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK949I WORK REGISTER SAME AS TRACE REGISTER

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK950I TRACE REGISTER OPERAND OMITTED

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK951I REGISTER ZERO INVALID AS LINK REGISTER

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK952I REGISTER ZERO INVALID AS WORK REGISTER

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK953I REGISTER ZERO INVALID AS TRACE REGISTER

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK954I INVALID CHARACTER PREFIX

Explanation: Self-explanatory.

User Response: Correct the prefix and reassemble.

BDK955I INVALID CODE SPECIFIED FOR OPERAND XXX

Explanation: The code specified on LTRTENT macro for operand XXX was invalid. The code was not within the range specified on the LTRTBEG macro or was not a self-defining term.

System Action: The translation table will still be generated, but this code will be defaulted.

User Response: Correct the error and reassemble if the generated table is unusable.

BDK956I INVALID CHARACTER STRING OPERAND XXX

Explanation: The OCHR operand was not specified as character or hexadecimal data, or the length of the data exceeded the maximum allowed, 7 for character and 8 for hexadecimal.

System Action: The translation table will still be generated but this code will be defaulted.

User Response: Correct the error and reassemble if the generated table is unusable.

BDK957I INVALID FUNCTION FOR OPERAND XXX

Explanation: The function code specified for operand XXX of the LTRTENT was not CAS1, CAS2, CAS3, CAS4, BSP, ADV, DS, TW, or user-defined value from X'01' to X'3F'.

System Action: The translation table will still be generated but this code will be defaulted.

User Response: Correct the error and reassemble if the generated table is unusable.

BDK958I INVALID ADDRESS FOR OPERAND XXX

Explanation: The Address specified in the LTRTENT macro for operand XXX exceeded 8 characters, or a mask operand was not specified.

System Action: The translation table will still be generated but this code will be defaulted.

User Response: Correct the error and reassemble if the generated table is unusable.

BDK959I INVALID INLEN FOR OPERAND XXX

Explanation: The value specified for the INLEN operand on the LTRTENT macro for operand XXX was not V, or within the range of 1 to 255.

System Action: The translation table will still be generated but this code will be defaulted.

User Response: Correct the error and reassemble if the generated table is unusable.

BDK960I OFF-POINTED STRING GT65535 OP=XXX

Explanation: The displacement value exceeded 65 535.

System Action: The translation table will still be generated but this code will be defaulted.

User Response: The table should be split into multiple tables and then reassembled.

BDK961I MASK FOR OPERAND xx IGNORED

Explanation: Invalid mask specified for the LTRTENT macro.

User Response: Correct the mask and reassemble.

BDK962I INVALID LOW RANGE FOR LTRTBEG

Explanation: The low range was not a self-defining term, or contained too many characters.

System Action: Default low range of 0 is used.

User Response: Correct the error and reassemble if the generated table is unusable.

BDK963I INVALID HIGH RANGE FOR LTRTBEG

Explanation: The high-range operand was less than the low range or it was not a self-defining term, or the operand contained too many characters.

System Action: Default of 255 is used.

User Response: Correct the error and reassemble if the generated table is unusable.

BDK964I INVALID MODE OPERAND ON LTRTBEG

Explanation: The mode operand was not ERROR, MOVE, SKIP, or a self-defined term.

System Action: Default of ERROR is used.

User Response: Correct the error and reassemble if the generated table is unusable.

BDK965I USAGE OF DEFCODE MACRO CONFLICTS WITH THE USAGE OF THE DEFLINK MACRO

Explanation: SEPASMB keywords are in conflict.

User Response: Correct the error and reassemble.

BDK966I ADDRESS 'table entry' IS TOO LONG

Explanation: Self-explanatory.

User Response: Correct the error and reassemble.

BDK967I TOO MANY OPERAND—EXCESS IGNORED

Explanation: This message usually occurs with another message indicating the failing instruction or operand, and may indicate that correct operands were dropped during assembly because of a syntax error.

User Response: Correct the specific errors indicated by other messages first and recheck the syntax of the affected instructions, then reassemble the program.

BDK968I CHNGDIR KEYWORD OPERAND INVALID

Explanation: Invalid CHNGDIR keyword option specified on LSEND.

User Response: Correct the error and reassemble.

BDK969I INVALID OPERAND (operand) FOR MACRO (macro)

Explanation: You specified an invalid keyword or positional operand on the named assembler instruction.

User Response: Correct the source, and reassemble.

BDK970I MISSING DEFSTOR SEGMENT SPECIFICATION

Explanation: You did not specify a required SEGSIZE segment value on the DEFSTOR instruction, or you tried to default to a predefined segment value by omitting the segment value rather than specifying zero (0).

User Response: Correct the SEGSIZE operand coding, and reassemble.

BDK971I INVALID REGISTER ADDRESS

Explanation: A normal or modified register address contains an incorrect segment, length, or displacement.

User Response: Correct the register address and reassemble.

BDK972I SINIT MACRO HAS NOT BEEN ISSUED

Explanation: An INITSEG or ENDINIT instruction appears in your program before or without an SINIT instruction.

User Response: One or more INITSEG instructions must begin with an SINIT instruction and end with ENDINIT, and have no other intervening instructions. Correct and reassemble your program.

BDK973I LENGTH 1 TOO LARGE—USE REGISTER FORM

Explanation: The length field assembled for operand 1 is too large for a nonregister address.

User Response: Recode the instruction using register addressing, which allows a length of up to 64K, depending on the instruction. Reassemble the program.

BDK974I LENGTH 2 TOO LARGE—USE REGISTER FORM

Explanation: The length field assembled for operand 2 is too large for a nonregister address.

User Response: Record the instruction using register addressing, which allows a length of up to 64K, depending on the instruction. Reassemble the program.

BDK975I IMPROPER ERROR CODE PRESENTED

Explanation: An error code occurred that has no corresponding error message.

User Response: Request assistance from your programming liaison or service facility.

BDK976I DEFSTOR IS REQUIRED WITH APENTRY

Explanation: APENTRY was specified on the BEGIN instruction, but a DEFSTOR instruction was not included in your program or the DEFSTOR does not precede the FINISH instruction.

User Response: Check your program, correct any error, and reassemble.

BDK977I END OF DEFSTOR EXCEEDS 4096

Explanation: The DEFSTOR instruction occurred partially or completely outside the first 4096 bytes of your program.

User Response: Recode the DEFSTOR instruction at an earlier point in your program, within the first 4060 bytes (address X'FDC', or earlier).

BDK978I SEGMENT (segment) LENGTH TOO LONG, MAXIMUM SIZE FORCED

Explanation: You defined a segment on the DEFSTOR instruction longer than 65 535 bytes.

System Action: A default of 65 535 is assumed.

User Response: Restructure your program, if necessary, to accommodate the segment limit.

BDK979I USE INVALID, DYNAMIC ASSUMED

Explanation: The DEFSTOR instruction's USE parameter does not specify STATIC or DYNAMIC.

System Action: DYNAMIC is assumed.

User Response: Recode the DEFSTOR instruction if you want the storage defined as STATIC.

BDK980I REDEF LABEL MUST REFERENCE SAME LEVEL

Explanation: The REDEF keyword of L1 - Ln instructions refer to previous L1 - Ln instructions but the levels are not equal.

User Response: Recode the instructions specifying the same level.

BDK981I LDSECT ENCOUNTERED BEFORE LEND

Explanation: Two or more LDSECT instructions without an intervening LEND.

User Response: Add LEND instruction and reassemble.

BDK982I REDEF AS label IS SMALLER THAN DEFN. xx FILLER BYTE(S) ADDED.

Explanation: The definition of a level field is smaller than the original specification. The field has been padded to the same length.

User Response: None

BDK983I REDEF AT label LABEL IS LARGER THAN DEFN STRUCTURE. EXPANSION TERMINATED.

Explanation: The redefinition of a level field is larger than the original field and the structure size cannot be determined.

User Response: Correct the redefinition and reassemble.

BDK984I NO LDSECT ENCOUNTERED BEFORE LEVEL DEFINITION.

Explanation: A level instruction (Ln) was coded before an LDSECT.

User Response: Add LDSECT and reassemble.

Appendix D. Program Check Codes

If the 4700 controller encounters an execution request that indicates a logic error, a program check results. The following are the hexadecimal codes and the explanations for possible program checks:

Code	Explanation
01	Invalid segment specification: An operand specifies a segment that was not defined during controller configuration procedure, or Segment 14 was specified in an instruction that will cause data to be stored or changed in Segment 14.
02	Segment overflow: Completion of the instruction requires more storage than the specified segment provides.
03	Field length error: An incorrect field was specified. The length is greater than 2 for an immediate operand; or a SETFPL instruction attempted to adjust the field length indicator to a negative value; or a value is specified which, when added to the PFP, would be greater than the segment length; or the field length was greater than 255 for a PAKSEG instruction.
04	Return-address stack error: An LRETURN instruction was issued, but the return-address stack was empty; or a branch instruction was issued, but the stack was full.
06	Instruction count threshold: The number of instruction executions allowed per transaction has been exceeded.
08	No overlay name: The overlay name is not in the resident overlay directory.
09	Invalid operation or segment code: The instruction operation or segment selection code specified is invalid. Make sure that any required OPTMOD coding for the instruction was entered and that any parameter fields are properly coded.
0A	No entry point: There is no startup entry point specified.
0B	Instruction address error: An addressing error has occurred. In the case of branch instructions, the program check address field of Segment 1 will contain the address of the branch instruction.
0C	Instruction count exceeded: 65,535 instructions have been executed without a release of control.
0D	DEFDEL missing or incorrectly used: Either a delimiter request was made but no delimiter table was found or the table is not halfword aligned.
0E	EDIT mask error: The mask used with an EDIT instruction contains an error.

0F Invalid link write control field: The link write control field or write options are invalid.

10 Communication link write length error: Data length exceeds 4095, data length during an LWRITE in batch mode was too long, command data length is incorrect; negative-response data length is incorrect, or there was a negative response to setting or testing sequence numbers.

11 Invalid parameter list, or parameter space is insufficient.

12 Indexing is not active.

20 Program check in called application program.

21 Called application program not found.

22 APCALL link stack full.

23 Recursive APCALL to an application program defined as USE=STATIC during configuration.

24 APCALL storage pool defined by MAXSTOR=was exceeded.

25 APCALL segment pool defined by MAXSEG=was exceeded.

26 APRETURN issued with no APCALL link stack entry - no calling application program.

27 Register address contains invalid segment space ID.

28 No transient pool: a transient pool was not defined for this station.

29 Transient application size error: The target transient application program will not fit in the largest transient area defined in the pool for this station.

FF System error.

Appendix E. Status Codes

The list below and the table that follows gives information about the two bytes of status bits that are set in SMSDST when an exceptional condition occurs (condition code=X'02'). The status bits in the first byte (SMSDS1) indicate the general condition:

Bits in SMSDS1		*Condition*
-------1	(X'01')	Incorrect length
------1-	(X'02')	Unit check
-----1--	(X'04')	Command reject
----1---	(X'08')	Attention
---1----	(X'10')	Prior operation
--1-----	(X'20')	Data check
-1------	(X'40')	Unit exception
1-------	(X'80')	Intervention required

The status bits in the second byte (together with those in the first) indicate the specific condition, as shown in the table.

The table applies to communications between stations. To use the table, find the status bits in the leftmost column of the table, the applicable instruction in the third column; read the explanation of the corresponding condition in the second column of the table.

A status value not in the table may be a combination of status codes. When such status values occur, review the table and search for the highest value first, then the next highest value. Remember that a status bit can be shared by more than one status code.

Status Bits	Condition	Instruction
-------1 -------1 (X'0101')	Incorrect length: The message was longer than the space available in the segment (that is, the space between the PFP and the end of the segment). Or the message was longer than the value of the FLI when the FLI was nonzero and less than or equal to the length between the PFP and the end of the segment. Or a write to a logical work station contained more than 255 bytes. *Action:*Change the segment so that enough space is available for the message. Or, if the end of the field was unexpected, change the FLI.	LREAD LWRITE
-----1-- -------1 (X'0401')	Command reject: No asynchronous station entry point defined in the current controller application program of the station receiving the message from the write operation. One of the following may may have caused the condition: the receiving or sending application program may have been incorrect; a message may have been sent to all stations whether they could receive it or not. *Action:* If necessary, change the application program of the receiving station so that it has an asynchronous station entry point or change the application program of the sending station so that no messages are sent to the other station.	LWRITE

Figure E-1 (Part 1 of 2). Status Codes

Status Bits	Condition	Instruction
-----1-- ------1- (X'0402')	Command reject: The station ID specified was not the ID of any station that exists for the controller application program sending the message. One of the following may have caused the condition: the station IDs may have been specified incorrectly during the controller configuration; the application program may have specified the station ID incorrectly; or a message may have been broadcast to all stations whether they existed or not. *Action:* If necessary, check the controller configuration and the controller application program and make corrections.	LWRITE LCHECK

Figure E-1 (Part 2 of 2). Status Codes

Status Bits	Condition	Instruction
----1--- -------- (X'0800')	Attention: The operator signaled attention by pressing the Reset key twice in succession. The operation was in a wait state with an indeterminate end point (an attention does not affect a wait state with a determinate end point). The wait state may have resulted from such condition as: a read from a 4704 terminal or the central processor; intervention required for a printer; failure to encode a magnetic stripe after the magnetic stripe encoder was enabled. *Action:* Prompt the operator to carry out the appropriate action (such as replacing the forms on a printer). Reset the magnetic stripe encoder if it was enabled.	LCHECK
-1------ -------- (X'4000')	Unit exception: No message was found for the read operation because no message was pending for the station. *Action:* Ignore the lack of a pending message.	LREAD
1------- -------- (X'8000')	Intervention required: A message was already pending for the station that was to receive the message from the write operation. *Action:* Wait until the pending message has been read.	LWRITE

Figure E-2. Status Codes

Appendix F. Functions Retained for 3600 Compatibility

Split Programs

The split program structure separates the instruction section of your program from the constants section, and is supported on the 4700 for compatibility with the IBM 3600 Finance Communication System. You must specify the split option during configuration. The split program structure can be used with overlay programming, and applies to either the relocatable or nonrelocatable programs described earlier.

The following paragraphs describe the operands of the instructions in Chapter 5 that control the split programming capability.

APOPT Instruction: SPLIT Operand

The split assembly option (SPLIT=Y specified in the APOPT instruction) produces two CSECTs. The OVLYSEC instruction label is used to name the constant section and an internally generated label is used to name the instruction section. If SPLIT=Y is specified, SECTION AUTO is assumed unless overridden by a SECTION instruction. The default is (N).

BEGIN Instruction

APBNM Operand

If SPLIT=Y option was specified in the APOPT instruction, two CSECTs will be generated; a CSECT with the name identified by APBNM=name for the constants and a CSECT with an internally created unique name for the instructions.

INSNAME Operand

A 1-to-8 character symbolic name to be associated with the instruction portion of a split application program. If this name is omitted, the internal name BQKINx is assigned, where *x* is an integer in the range 1-999.

DEFCON Instruction

When SPLIT=Y and RELOC=N are both coded, (in the APOPT instruction) and a label in the instruction section of the program is referred to in the (LABEL-APBNAME) format, the format must be changed to:

- (label-BQKIN1), where BQKIN1 is the CSECT name of the instruction portion
- or the (label-name) format where *name* is the symbolic name assigned to the instruction section by the INSNAME operand of the BEGIN instruction.

LEXEC Instruction

When the program is assembled using the SPLIT option, the addressed instruction must be in an instruction section, if the addressed instruction is in a constant section, the results are unpredictable.

LLOAD Instruction

When you use the LLOAD instruction to load a split overlay containing a constants section only, set the ELPCSF flag and the ELPSEG and ELPCLA parameters. Do not set the ELPISF flag. For a split overlay containing an instruction section only, set the ELPISF flag and the ELPILA address. Do not set the ELPCSF flag. In either case, setting both flags causes program check (hex 17), invalid parameter list.

ELPCSF
: If the SPLIT option is specified and this flag is on in ELPFLG then the LLOAD instruction derives the load address for the constants from the ELPSEG and ELPCLA parameter list fields.

ELPISF
: If the SPLIT option is specified and this flag is on in the ELPFLG field then LLOAD obtains the load address for the instructions from the ELPILA parameter list field.

LSEEK Instruction

The table must be located in the constants section of a program assembled with the SPLIT=Y option in the APOPT instruction. (Use the LSEEKP instruction to search a table located in the instruction portion.)

LSEEKP Instruction

If SPLIT=Y is coded in the APOPT instruction and the table is in the instruction portion of a split program, the PFP and FLI are not changed. LSEEKP searches tables in either the constants or instruction section of a split or non-split application program.

You must set the following parameter list fields before issuing the LSEEKP instruction.

SKPFLG1 A 1-byte flag field.

Hex Code	Meaning
xx0x xxxx	The table is in a non-split program or in the constants portion of a split program.
xx1x xxxx	The table is in the instruction portion of a split program.

LSEEKPL Instruction

NOINST

Specifies that the table to be searched is located in a nonsplit program, in the constant portion of a split program, or in any segment.

INST

Specifies that the table to be searched is located in the instruction portion of a split program.

OVLYSEC Instruction

OVLYSEC defines the load addresses for the instruction section and the constant section of an overlay assembled with the split option (SPLIT=Y in the APOPT instruction).

inst-org/const-org Operand

The *inst-org/const-org* operand is used to specify the load address for the instruction and constant sections, respectively, of an overlay when assembling with the split option in effect. One of these parameters must be specified. If one is omitted, the omitted operand will default to an '*' specification.

INSTR
Specifies the presence (Y) or absence (N) of an instruction portion in the overlay section. This applies only when SPLIT=Y is specified on the APOPT instruction. The default is Y.

INSNAME
A 1-to-8 character symbolic name to be associated with the instruction portion of a split application program. If this name is not supplied, the instruction portion is given the name BQKINx, where *x* is in the range 1-999.

SECTION Instruction

INSTR
Specifies that following instructions are to be placed in the instruction section. INSTR works only if SPLIT=Y has been specified.

CONST
Specifies that following instructions are to be placed in the constant section. CONST works only if SPLIT=Y has been specified.

AUTO
Causes an automatic collection of instructions and constants into respective CSECTs. AUTO works only if SPLIT=Y has been specified.

SEGCODE Instruction

For the SPLIT=Y option, two CSECTs will be generated: one with the name specified in SEGCODE for the constants and another with an internally generated name for the instructions.

INSNAME Operand: The INSNAME operand is a 1-to 8-character symbolic name to be associated with the instruction portion of a split application program. If this name is not supplied, the instruction portion is given the name BQKINx, where *x* is in the range 1-999.

SEGCOPY Parameter List

Byte 1 - Segment/Section Indicator: If the second segment number is 14, this field can be specified as a character C or I, to identify whether the constant or instruction section of the segment is being referenced. Unless a value of I is coded, C is assumed. If the application program occupying Segment 14 for the specified station was assembled without the SPLIT=Y option, this field is ignored.

STOVLY Instruction

C

This STOVLY identifies the origin address of the constant portion of a split application overlay.

I

This STOVLY identifies the origin address of the instruction portion of a split application overlay.

A

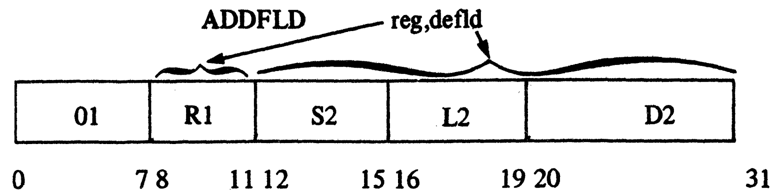
The preceding SECTION instruction controls whether this STOVLY identifies the instruction or constant portion of a split application overlay. The default is A.

TABLE Instruction

An LSEEK table must be in Segment 14 (for a split program, the constants section). A table for LSEEKP can be in any part of the program. The SRT operand is used with the LSEEKP instruction to provide a binary search of a sorted table.

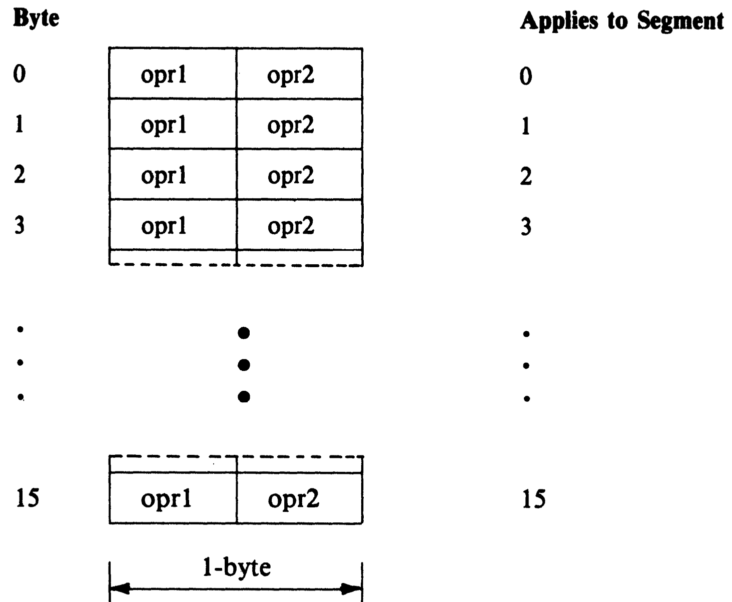
Segment Indexing

Segment indexing is the ability to store a displacement value in a register and have this displacement value added to the displacement (address) of data referenced in a machine instruction. For example, consider the machine instruction:



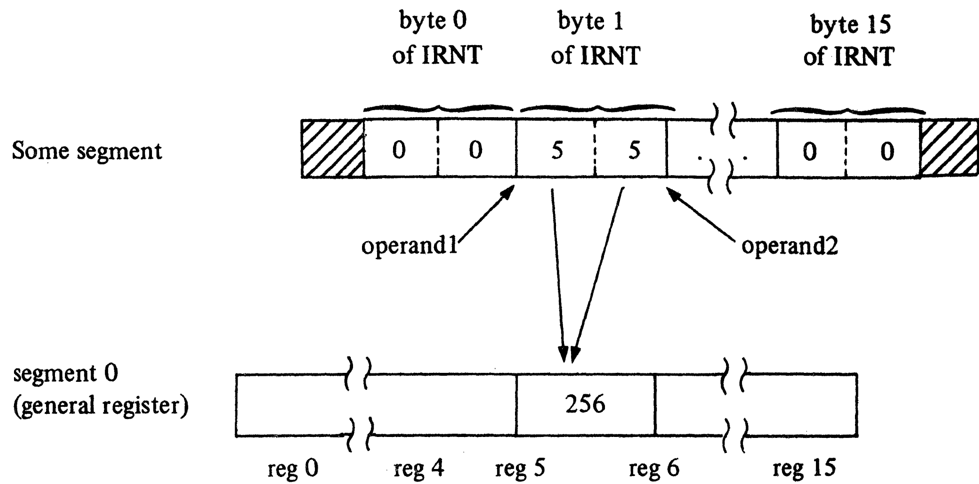
Without segment indexing, D2 points directly to the data within Segment S2 that is to be added to R1. With segment indexing, an index value is added to the displacement in D2 to locate the data within Segment S2. The index value is a binary number from 0 to 65 535 and is located in the low-order 2 bytes of any of the general registers from 1 to 15. Register 0 has a special meaning and is discussed later. The use of indexing allows real-time addressing of data during program execution.

Segment indexing is applicable to all instructions that use fixed-field addressing with the exception of the SETFPL and SETSPF instructions. Before using segment indexing, an application program must first define a 16-byte Index Register Number Table (IRNT) in any segment except Segments 0 and 14. Each 1-byte field of this table corresponds to a segment, with byte 0 corresponding to Segment 0, byte 1 to Segment 1, and so on, as follows:



Because the instructions that use indexing can have either one or two fixed-field operands, each byte of the IRNT contains two 4-bit entries for the application segment. The first four bits refer to operand1 of an instruction, and the second four bits refer to operand2. (FF instructions have both operand1 and operand2. FI instructions have only operand1. RF and SF instructions have only operand2.) Each four bits represent a register number from 1 to 15, and the specified register contains the actual index value. If a 0 is specified as a four-bit entry in the IRNT, then the index value for that operand (opr1 or opr2) referring to that particular segment is 0, regardless of the contents of register 0.

In the following example, instructions with fixed-field operands that refer to data in Segment 1 will have the value in register 5 (a decimal 256) added to the displacement for both operand1 and operand2 of the machine instruction.



The purpose of two register numbers per segment is to simplify simultaneous index addressing of two different sets of DEFLDs (for example, two different record formats) within one segment. For simplicity, if the user requires an application with simultaneous index addressing of two different sets of DEFLDs, place each set in a different segment. Then, the program is not concerned with the operand1/operand2 distinction because only one index register is used per segment, as in the above example.

A record of the indexing status for each station and segment is maintained internally by the controller and determines whether indexing is active or inactive for the station and segment. Indexing is initially inactive for all stations. Each station requiring indexing must execute a SETX ON instruction to activate indexing for that station, to define the location of the IRNT, and to cause the controller to record the active/inactive status for each segment that the station can access. The indexing status for Segments 13, 14, and 15 is unique for each logical work station. If two Segment 0s exist for one station, and if indexing is set active (or inactive) for one Segment 0, it is actually set active (or inactive) for both.

The SETX ON instruction can also be used to cause an indexing active station to change from one IRNT to another. The controller's indexing active/inactive status is updated for the new IRNT contents.

The SETX OFF instruction is used to deactivate segment indexing for the current station.

When indexing is active for a station, the SETXREG is used to modify an active IRNT. SETXREG updates the segment's IRNT and also updates the active/inactive status as recorded by the controller. Using instructions such as STFLD and MVFXD to modify an IRNT entry does not update the controller's recorded status for the segment.

If a segment does not require indexing but has been activated with a SETX ON instruction, it is generally better to change the entry in the IRNT for that segment to X'00' than to have indexing active for the segment with an index value of zero in the indexing register. Both approaches would cause the relevant fixed-field operands to address the same data; however, address resolution with an index value of zero is slightly slower.

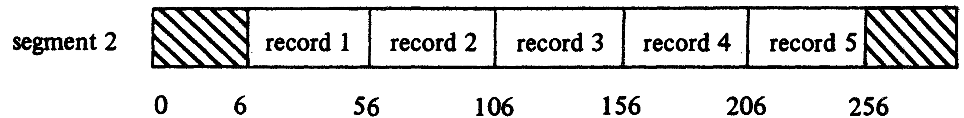
The TESTX instruction is used to test whether indexing is active for a specified station or segment.

The following example illustrates how an application program might take advantage of segment indexing to refer to fields in a sequence of similar records. The example assumes that indexing is active (the SETX ON instruction has been issued) for the station and the IRNT entry for Segment 2 is X'33'. Each record is 50 bytes long and contains information describing a customer's checking account, as follows:

Displacement
into Segment

```
0          ACTNAM   DEFLD    2,0,20   NAME
20         ACTNUM   DEFLD    2,,9     NUMBER
29         ACTFLG   DEFLD    2,,2     FLAGS:
           ACTFMN   EQUATE   X'0001'    ON   -- No minimum balance
                                               required
           ACTFOV   EQUATE   X'0002'    ON   -- Eligible for
                                               overdraft
31         ACTBAL   DEFLD    2,,4     CURRENT BALANCE
                    .
                    .
                    .
```

Assume that 5 of the above records are consecutively located in Segment 2, with the first record beginning 6 bytes into the segment. (The records might be part of a disk/diskette data set where each 256-byte block contains 5 records.)



The following application computes a total of the current balances for those accounts that are eligible for overdraft. The IRNT indicates that register 3 is Segment 2's index register. Upon completion, register 4 will contain the desired total.

```
H000  DEFCON   H'0'
H006  DEFCON   H'6'
H050  DEFCON   H'50'
H256  DEFCON   H'256'
       .
       .
       .
      LDFLD    3,H006     Load index register with address
                            of record 1
      LDFLD    4,H000     Initialize total to zero
LOOP  TSTMSKI  ACTFLG,
               ACTFOV     Is account eligible for overdraft          1
      JUMP     MZ,NEXT      NO  —then go past for next record
      ADDFLD   4,ACTBAL     Yes —add current balance to
                                 total                               2
NEXT  ADDFLD   3,H050     Add record length to index register
      CAFLD    3,H256     Are there more records
      JUMP     LT,LOOP      Yes —go check next record
       .                    No  —desired total is in
       .                             register 4                      3
       .
```

Notes:

1. This instruction will be executed 5 times, once for each record. The following table shows addresses (with respect to Segment 2) actually referenced by the field ACTFLG on successive execution.

Execution Number	Displacement (in TSTMSKI's Machine Instruction)	+	Index Value (in register 3)	=	Address Actually Referenced
1	29		6		35
2	29		56		85
3	29		106		135
4	29		156		185
5	29		206		235

2. Because ACTBAL references Segment 2, it also will be indexed.

3. To further reduce application program execution time and size, the TSTMSKI and JUMP instructions could be replaced with a LIFOFF instruction, and the ADDFLD, CAFLD, and JUMP instructions could be replaced with a BRANX instruction.

Indexing Affects on Instructions

This section describes how indexing affects certain instructions.

BRANX Instruction

Because the instruction algebraically increases the low-order 2 bytes of a register, BRANX can also be used as an aid to simplify table processing.

LDRA Instruction

If indexing is active in the specified segment, LDRA adds the index value to the specified displacement before it is placed in the operand 1 register. When referenced during program operation, a register address is unaffected by indexing.

MVDI Instruction

If the fixed field is indexed, then the new setting of the primary field pointer will include the index value.

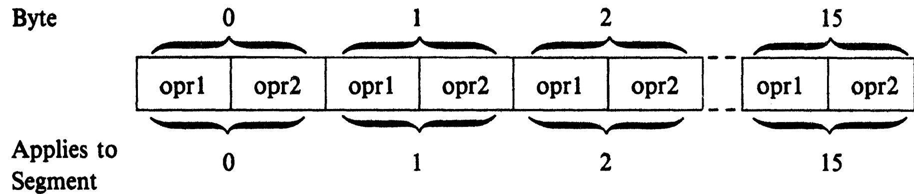
MVFXD Instruction

If the fixed field is indexed, then the new setting of the primary field pointer will include the index value.

SETX--Enable/Disable Segment Indexing

The SETX instruction activates or deactivates segment indexing for the current station. When the first operand is ON, indexing is activated for this station and the location of the Index Register Number Table (IRNT) is also specified. When the first operand is OFF, indexing is deactivated for this station and no reference is made to the IRNT. Register- or modified register-addressed operands cannot use indexing.

The IRNT is 16 bytes long and resides in any segment except 0 or 14. It has the following format:



Because the instructions that use indexing can have either one or two fixed-field operands, each byte of the IRNT contains two 4-bit entries for the applicable segment. The first four bits refer to operand 1 of an instruction, and the second four bits refer to operand 2 (FF instructions have both operand 1 and operand 2, FI instructions have only operand 1, and RF and SF instructions have only operand 2). Each four bits represent a register number from 1 to 15, and the register contains the actual index value. If a 0 is specified as a 4-bit entry in the IRNT, indexing is not used for that operand.

Note: To use this instruction, you must code the P68 operand of the OPTMOD configuration instruction.

Name	Operation	Operand
[label]	SETX	{ OFF|ON } , { defld2 | (defrf2) | (reg2) | seg2,disp2 }

ON

Indexing is to be activated for this station.

OFF

Indexing is to be deactivated for this station.

operand 2

Is the label of a DEFLD instruction which defines the location of an IRNT. The implied length is ignored; the table is assumed to be 16 bytes long. The segment number associated with this table cannot be 0 or 14. If a register-addressed IRNT is specified, the segment space ID in the register must equal the current segment space ID.

Condition Code: The code is not changed.

Program Checks (hex): 01, 02, 09, or 27 can be set.

SETXREG--Set Index Register Number

The SETXREG instruction modifies the contents of the IRNT for a station that is active for indexing. See the SETX instruction to activate indexing for a station. See SETX for a description of the IRNT.

Note: To use this instruction, you must code the P68 operand in the OPTMOD configuration macro.

Name	Operation	Operand
[label]	SETXREG	{ (seg1,[reg1]) | (seg2[,reg2]) } [, { (seg2[,reg2]) | DUP }]

seg1

Locates the byte in the IRNT that corresponds to seg1. If seg1 is specified, the leftmost four bits of the located byte are set to reg1. If seg1 is omitted, no change is made to the leftmost four bits in any byte of the IRNT.

reg1

Is a decimal integer from 0 to 15. Integers 1-15 indicate a register number. If seg1 is specified, the default for reg1 is 0. When reg1 is 0, the index value is zero regardless of the contents of reg0.

seg2

Locates the byte in the IRNT that corresponds to seg2. If seg2 is specified, the rightmost four bits of the located byte are set to reg2. If seg2 is omitted, no change is made to the rightmost four bits in any byte of the IRNT (unless DUP is coded).

reg2

Is a decimal integer from 0 to 15. Integers 1-15 indicate a register number. The default for *reg2* is 0, if seg2 was specified. If reg2 is 0, the index value is zero regardless of the contents of reg0.

DUP

Is an abbreviation that will provide seg2,reg2 with the same values given to seg1,reg1. DUP is valid only when operand 1 is specified.

Condition Code: The code is not changed.

Program Checks (hex): 01, 09, or 12 (indexing inactive) may be set.

TESTX--Test for Active Indexing

The TESTX instruction tests the indexing status (active or inactive) for a station or a specified station segment. The instruction points to a parameter list (see COPY DEFTSX). A flag byte (TSXFLG) in the parameter list must be set list (see COPY DEFTSX). A flag byte (TSXFLG) in the parameter list must be set to X'00' or X'80' as follows:

TSXFLG=00 (test indexing status of station)

To test whether indexing is active for a specified station, the program must first initialize the TSXFLG field to X'00' and enter a station number in the TSXSID field of the parameter list. If indexing is inactive for the specified station, the parameter list is unchanged.

If indexing is active for a specified station, the TSXFLG field is set to X'02' and the segment number and address of the IRNT are also returned in the parameter list in fields TSXSEG and TSXADR, respectively.

TSXFLG=80 (test indexing status of station and segment)

To test whether indexing is active for a specified station and a specified segment, the program must first initialize the TSXFLG field to X'80', enter a station number in the TSXSID field, and enter a segment number in the TSXSEG field of the parameter list.

If indexing is inactive for the specified station, the parameter list is unchanged.

If indexing is active for the specified station, the two register numbers from the IRNT for the specified segment are returned in the parameter list in fields TSXREG1 and TSXREG2. TSXFLG is set to X'82' if the segment is not active for indexing or to X'83' if the segment is active for indexing.

Testing for active/inactive indexing for the segment is via the controller's internal indicators which are set/reset through SETX and SETXREG instructions.

If the TESTX request is not valid (for example, an invalid station). TESTX will not modify any part of the parameter list.

Note: To use this instruction, you must code the P68 operand on the OPTMOD configuration instruction.

Name	Operation	Operand
[label]	TESTX	{ defld2 | (defrf2) | (reg2) | seg2,disp2 }

operand 2
Defines the field that contains the parameter list. The implied length is ignored; the parameter list is fixed in length. The segment number for this operand cannot be 14.

Condition Code: One of the following is set:

Hex Code	*Explanation*
01 OK	TESTX was successful
02 ID	Invalid station ID in the parameter list
04 IS	Invalid segment number in the parameter list (can only be returned if a segment's indexing status is being tested).

Program Checks (hex): 01, 02, 09, 11, or 27 can be set.

Appendix G. Program Communication with the System Monitor

Application Program Debugging

You can use the control operator's console to debug controller application programs. Any application program may be accessed as it operates on behalf of one of its associated logical work stations. The following may be done when in debugging mode:

- *Address stop*: The controller application program can be stopped just before an instruction is executed so that station storage or global storage can be examined.

- *Display*: Any location in programmable storage, station storage, global storage, or the controller application program may be displayed on the 4704/3278.

- *Alter*: Any location in programmable storage, global storage, or the controller application program may be altered by entering the hexadecimal equivalents of the bytes to be altered. The storage or application program is altered only for the current day's operations; a permanent change requires that the application programs or configuration be changed by creating a new operating diskette. Because anyone who knows the system monitor logon procedure and the operator password can alter an application program, you should be careful to protect the password.

  The application program on the diskette may also be altered using the 032 command.

  Note: Any alter command, including 032, will not be accepted if the diskette has been declared operational during configuration assembly (DSKOP=YES).

- *Instruction step*: The controller application program can be stopped after each instruction is executed. The addresses of the instructions and 8 bytes at those addresses are displayed in hexadecimal when the controller stops. To execute the next instruction, the EOM key is pressed.

- *Hard-copy trace*: The order of execution of the instructions in a controller application program can be traced. Each instruction address and 8 bytes at that address are printed on the specified hard-copy terminal (that is, the journal and administrative printers). *Hard-copy trace* differs from *instruction step* in that the controller does not stop after each instruction is executed.

- *Checking indexing status*: If segment indexing is being used, its status may be checked using command 15.

Programmable Input Facility

The programmable input facility of the system monitor enables a logical work station to input commands to the system monitor. These commands could be stored in disk or diskette data sets, or could be received from a host program communicating with the logical work station. A controller application program could be written specifically to support logical work station control of the monitor,

or routines could be incorporated into programs doing other processing also. The program issues LREADs and LWRITEs to station 1 (SMSDSS=1) to communicate with the monitor. A flag in the GMSFLG field (GMSFPCM) is set on by the controller while the system monitor is processing the command. This flag is set to 0 when command processing is complete.

If the work station is also in communication with a host application program that accepts input from a host terminal or if the Communications Network Management/Controller Support (CNM/CS) facility is installed on your system, a host terminal can become the system monitor terminal for any attached 4700 controller.

When not in communication with a work station or being controlled by a remote operator at the host, the system monitor can still be controlled from a 4704/3278 keyboard display with all commands valid.

Monitor Restrictions under Programmable Input Control

When the programmable input facility is being used, some system monitor commands are limited or not allowed. The following list notes the differences.

1. If the 049 command had been issued the line value that was entered is ignored except on the 001 command. Each response will contain the full message normally displayed.

2. Debugging mode is not allowed for the station that is currently communicating with the system monitor via station reads and writes. If a Debug request (123) for that station is sent, a 90013 error message will be returned.

3. Logon (three Attentions plus the monitor ID) is not required when using the programmable input facility. Any write (valid or not) to station 1 will serve as the logon, provided that no one else is logged on as the monitor. When you are logged on in programmable input mode, only the 000 command (if in Debug mode, must be 00, then 000) or a station read or write error will log off the monitor and make it available for another station, CNM, or 4704 terminal. A 90032 message will be sent if a station read or write error occurs.

4. If a station attempts to communicate with the system monitor while it is logged on to another station, the request will be rejected and a 90033 error message returned.

   Note: If the system monitor is currently logged on to a 4704/3278 terminal, the message will not be read until a logoff (000) is issued at the 4704/3278. A flag in GMSIND field (GMSISMM) is set on whenever a 4704/3278 is logged on to the system monitor. To avert a possible lockout, this flag should be checked before a write operation is requested from the monitor.

5. After a station writes a command to the system Monitor it should check for the LWRITE completion. This check should be in the form of a pause loop that includes the following instruction.

```
LCHECK ST,TIO
```

   In the pause loop you should check the SMSAFL flag for appropriate interrupts. Waiting for interrupts in this manner will prevent possible lockouts where the station and the System Monitor attempt to write to each other at the same time.

6. A zero length write to station 1 is treated as an Attention or Enter key. The action taken by the system monitor depends on the function it is executing at the time it reads the zero length write. If an attention was expected, the zero length write is accepted; however, a 90001 error message is returned. You should ignore the error message. If both an Attention and Enter key can be accepted, the system monitor will always assume that the enter key was the response.

7. During create disk (999) or transmit (888), any station write by the station logged on to the system monitor will be interpreted as an attention after communication with the host has started. If a read CPU is outstanding, the read will break with an attention error (0800) or prompt message 00020 or 00090.

8. While in programmable input mode, pressing the Attention key 15 times in 2 minutes on any idle 4704 or 3278 will log off the system monitor. Three more attentions should activate the monitor at that 4704/3278 and lock out the programmable input facility. If, at the time the 15 attentions are given, the monitor is in create disk (999) mode, logoff will not take place; instead the attentions will be ignored.

9. If create disk (999) is going to be used, the controller configuration procedure must specify at least one read buffer of 256 bytes or more (CNL operand on the COMLINK macro). If the expanded system monitor with multiple-block support is used (MONITOR=EXPMB in the STARTGEN macro), more than one read buffer should be specified. The number of required read buffers is proportional to the speed of the communications line. (Enough buffers should be specified so that, when a diskette write operation is in progress, data from the host can continue to be accumulated.) Two read buffers are usually sufficient for line speeds up to 4800 bits per second. To realize any performance benefits from the expanded system monitor's support of multiple-block I/O, communication-line speeds of at least 7200 bits per second must be used.

Bibliography

The publications listed below contain information that may be useful to persons programming a 4700 system.

IBM Vocabulary for Data Processing Telecommunication and Office Systems, GC20-1699

IBM System/370 Bibliography, GC20-0001

IBM System/370 Bibliography of Industry Systems and Application Programs, GC20-0370

IBM 4700 Finance Communication System: System Summary, GC31-2016

Subsystem Operating Procedures, GC31-2032

Subsystem Problem Determination Guide, GC31-2033

Host Support User's Guide, SC31-0020

4701 Controller Operating Instructions, GC31-2022

4704 Display Operating Instructions, GC31-2025

Systems Network Architecture (SNA) General Information, GA27-3102

Index

Special Characters

(defrf) 4-5
(defrf) addressing, use and example of 2-17
(reg) 4-4
"Allocating" storage 2-7
"Defining" storage 2-7

A

ACP operand 2-24
active logical work station, definition of 2-23
added function for release 3 vii
ADDFLD 2-17
ADDFLD and ADDREG, using 3-11
ADDFLD–Add Field 5-3
ADDFLDL–Add Field Logical 5-5
ADDMEM 5-27
ADDREG–Add Register 5-7
address constants, AL2 and YL2 3-1
Address stop G-1
addressing between primary and secondary programs 2-13
addressing data 2-1
addressing storage and register locations 2-13
addressing, base 3-3
addressing, modified register 2-17
addressing, register 2-16
ADDZ–Add Zoned Decimal 5-9
ADDZ, using 3-12
ADRLST–Return Address List 5-11
    IRETURN 5-11
alert messages, program check CNM 2-27
alter, storage G-1
AL2 address constants 3-1
ampersand 4-5
AND–AND Field 5-13
AND, using 3-12
ANDI–AND Field Immediate 5-15
ANDI, using 3-12
APBDUMP 3-18
APBDUMP–DUMP Segment or File to Diskette 5-17
    DEFDMP 5-17
APCALL 2-3
APCALL and APRETURN, using 3-14
APCALL–Call Assembler Application Program 5-19
    BEGIN 5-19
    DEFSTOR 5-19
APCALL/APRETURN 3-2
APCALL, using 2-29
API operand 2-25
APLIST 5-19
APOPT 2-2, 3-1, 5-153
APOPT–Application Program Options 5-23
APOPT, using 3-1
application program 1-5, 2-7
application program header 2-2
application program name 2-27
application programs 2-1, 2-6
APRETURN–Return to Calling Program 5-25
    APCALL 5-25
    DEFSTOR 5-25
arithmetic 1-5
arithmetic/logical instructions 3-10
arithmetic, types of 3-10
assembler language 1-3
assembly control instructions, using 3-3
AST operand 2-25
asynchronous interrupts 5-21
asynchronous interrupts, entry point priority of 2-24
asynchronous requests 2-25
ATD operand 2-24
ATM operand 2-25

B

base addressing registers, using 3-3
BEGIN 2-2
BEGIN–Assembly Control 5-27
BEGIN, detailed description 3-2
BEGIN, using 3-1
bibliography X-1
binary 4-5
binary arithmetic 3-10
binary operations 3-11
Binary Synchronous Control (BSC3) 1-2
bit control with LSETON and LSETOFF 3-13
bit-by-bit testing 3-13
braces 4-1
brackets 4-1
BRAMLR 2-4
BRAN 3-14, 3-15
BRAN–Branch 5-31
    APOPT 5-31
    EQUATE 5-31
Branch Instructions 3-14
branching, condition code 3-14
BRANL 2-4, 3-14, 3-16
BRANL–Branch and Link 5-33
    BRANLR 5-33
    BRANR 5-33
    LRETURN 5-33
    LSEEKP 5-33
BRANLR 3-14, 3-16
BRANR 3-14, 3-15
BRANR–Branch Register 5-37
BRANX 3-16
BRANX–Branch on Index 5-39
    APOPT 5-39

C

CAFLD–Compare Arithmetic Field 5-41
CAFLD, using 3-12
CAFLDL–Compare Arithmetic Field Logical 5-43
call programming 2-1, 2-3
calling programs, instructions for 3-14
calls 2-2
CAREG–Compare Arithmetic Register 5-45
CAREG, using 3-12
CCDI–Compare Character Data Immediate 5-47
CCDI, using 3-12
CCFLD–Compare Character Field 5-49
CCFLD, using 3-12
CCFXD–Compare Character Fixed 5-51
CCFXD, using 3-12
CCITT Recommendation X.21 1-2
ccmask 4-6
CCSEG–Compare Character Segment 5-53
CCSEG, using 3-12

chain dispatching, work station 2-23
changes/additions, release 3 vii
character 4-5
character, delimiter 2-20
characters, master compaction 3-8
check codes D-1
    APCALL D-2
    APRETURN D-2
    DEFDEL D-1
    EDIT D-1
    LRETURN D-1
    LWRITE D-2
    PAKSEG D-1
    SETFPL D-1
checking indexing status G-1
CNM (communications network management) 2-27
CNM/CS 5-145
CNM/CS. 5-146
    LREAD CP 5-146
    LWRITE CP 5-146
COBLCALL 2-3
COBLCALL–Call a COBOL Application Program 5-55
    APCALL 5-55
COBLCALL, destroying register 12 contents by 2-29
COBLCALL, using 3-14
COBOL 2-3, 5-129
COBOL (COommon Business Oriented Language) 1-3
COBOL programs, linking your program to 2-29
coding and syntax rules 1-5
coding rules 4-1
Communication Network Management/Controller Support (CNM/CS) G-2
communications network management (CNM) 2-27
COMP 3-8
COMP–Compress and Compact 5-57
compaction and compression, data 3-8
comparing data, binary and logical 3-12
compatibility, 3600 program 2-29
COMPTB–Build Compaction Table 5-61
COMPTB, using 3-8
COMPZ–Compare Zoned Decimal 5-65
COMPZ, using 3-12
condition code branching instructions 3-14
condition codes 2-27
configuration 1-3
constants 2-7
constants, AL2 and YL2 address 3-1
constants, defining 3-4
control instructions, assembly listing 3-4
control instructions, using assembly 3-3
control operator 1-3
control, ending work station 2-25
control, getting processing 2-24
control, instructions for program 3-14
control, releasing 3-14
controller timer 2-25
conversion, binary/zoned decimal 3-10
converting between decimal and binary data 3-7
COPY 2-5, 3-3, 4-6
COPY DEFCPL, using 3-9
COPY DEFDCP, using 3-10
COPY DEFTRP, using 3-6
copy files 2-1, 2-5
COPY files and lists (DEFxxx) 2-26
COPY–Copy Source Code 5-67
copyfilename 5-67
CPGEN iii
CPGEN usage limits, 3600 2-29
CPLTBD and CPLTBS fields of DEFCPL 3-9
CRETN–Conditional Return (COBOL) 5-69
CRETN, using 2-29

D

data 4-5
data comparing, binary and logical 3-12
data compression and compaction 3-8
data decompression/decompaction 3-10
data definition instructions 3-4
data movement 3-5
data operation instructions, using 3-5
data operations 1-5
data translation 3-6
data verification and checking 3-6
data, formatting input 3-5
debug G-1
debugging G-1
decimal arithmetic, zoned 3-10
DECOMP–Decompress and Decompact 5-71
DECOMP, using 3-10
decompression/decompaction, data 3-10
DECOMPTB–Build a Decompaction Table 5-75
DECOMPTB, using 3-10
DEFAPB 2-26, B-3
default 4-2
DEFCON 4-4
DEFCON–Define Constant 5-77
DEFCON, operand addressing using 2-13
DEFCON, table definition with 3-7
DEFCON, using 3-4
DEFCPL 5-57, B-5
    COMPTB 5-57
DEFDCP 5-71, B-5
DEFDCP, using COPY 3-10
DEFDEL–Define Delimiters 5-79
DEFDEL, using 2-20, 2-22, 3-4
DEFDMP–Define APBDUMP Buffer 5-83
    APBDUMP 5-83
DEFDMP, using 3-4
DEFELP 5-202, B-6
    DEFCON 5-204
    DEFLD 5-204
    OVLYSEC 5-204
DEFESP 5-371, B-6
DEFFAP 5-133, B-7
DEFGMS 5-67, B-7
DEFINT B-13
defintion instructions, data 3-4
DEFLD 4-4
DEFLD instruction, creating a 2-15
DEFLD–Define Field 5-85
    APOPT 5-85
    DEFCON 5-85, 5-86
    DEFLD 5-86
    DEFRF 5-85
    EQUATE 5-85
DEFLD, example of using 2-14
DEFLD, operand addressing using 2-13
DEFLD, using 3-5
DEFMER 5-205, B-14
DEFREG 2-26, B-14
DEFRF 2-6
DEFRF–Define a Modified Register Address Field 5-89
DEFRF, operand addressing using 2-13
DEFRGS 2-26, B-15
DEFSCA 5-321, B-15
DEFSCA, using COPY 3-5
DEFSCP 5-327, B-15
DEFSEG B-16

DEFSKP 5-226, 5-229, B-17
DEFSMS 5-17, 5-67, B-18
DEFSOR 5-237, B-26
DEFSTOR 2-17, 3-17
DEFSTOR–Define Segment Storage 5-93
DEFTRP 5-253, B-26
DEFTRP, using COPY 3-6
DEFTRT B-27
DEFTSX B-27, F-15
DEFVUE 5-411, B-28
DEFxxx COPY files and lists 2-26
delimiter table, defining location of a 3-2
delimiters 5-79
delimiters, defining and using field 2-22
delimiters, defining fields with 2-20
directory naming 3-1
DIRNAME= operand of APOPT 3-1
disp 4-4
DISP= operand of APOPT 3-1
dispatch control, detailed description of acquiring 2-24
dispatch tables, definition and limit 2-23
dispatches 1-5
dispatching cycle 2-23
dispatching logical work stations 2-23
dispatching, definition of station 2-23
displacement 2-13
displacement, definition of segment 2-13
displacement, use in modified register address 2-18
displacement, use in register addressing 2-17
display, storage G-1
DIVFLD and DIVREG, using 3-11
DIVFLD–Divide Field 5-95
    DIVFLDL–Divide Field Logical 5-97
DIVREG–Divide Register 5-99
DIVZ–Divide Zoned Decimal 5-101
DIVZ, using 3-12
dollar-sign insertion 5-107
    APOPT 5-108
DSECTs, defining 3-3
DTACCESS 3-17
DTACCESS–Data Access 5-103
DTAFREE 3-17
DTAFREE–Data Free 5-105
DUMMY SECTION 2-6
Dummy Sections 2-6
dummy sections (DSECTs), defining 3-3
dump 3-18
dump area, defining a 3-4

E

EDAM 5-17
EDIT–Edit Monetary Field 5-107
    MASK 5-107
EDIT, using 3-4
ellipsis 4-2
ending a DSECT (dummy section) 3-3
ENDINIT–End Initialization Section 5-111
    ENDINIT 5-111
    INITSEG 5-111
    SINIT 5-111
ENDINIT, using 2-12
ENDOVLY 2-2, 2-4
ENDOVLY–End of Overlay Section 5-113
ENDOVLY, using 3-3
ENDSEG 2-2
ENDSEG–End Application Program Section 5-115
    SEGCODE 5-115
ENDSEG, using 3-1, 3-3
ENTRY 2-6
entry point 2-5
entry points and priorities 2-24
entry points, defining and processing program 3-2
EQUATE 4-8
EQUATE–Equate a Label to a Value 5-117
    DEFCON 5-117
    DEFLD 5-117
    DEFRF 5-117
EQUATE, using 3-3
equating values 3-3
ERRLOG–Obtain Statistical Counters 5-119
error messages 1-5
error, entry point processing 3-2
errors, processing program check and condition
  code-related 2-27
example of defining fields within messages 2-22
example of moving fixed-length data 2-14
example of multiple-register-set programming 2-12
example, changing register address displacement 2-17
exclusive control 2-12
EXOR–Exclusive OR 5-123
EXOR, using 3-12
EXORI–Exclusive OR Immediate 5-125
EXORI, using 3-12
EXPS 2-16
EXPS–Exchange Primary and Secondary Field Pointers 5-127
    EXPS 5-127
Extended Disk and diskette Access Method (EDAM) 3-18
external label 2-6
EXTRN 2-6

F

F, +F, and -F
  use for variable-field addressing 2-20
FCLENTER and FCLEXIT, using 3-14
FCLENTER–Define COBOL Entry Linkage 5-129
FCLENTER, using 2-29
FCLEXIT–Define COBOL Exit Linkage 5-131
    APRETURN 5-131
    FCLENTER 5-131
FCLEXIT, using 2-29
field delimiters, defining and using 2-22
field length indicator (FLI) 2-15
field length, use in register addressing 2-17
field, length 2-18
fields, defining message 2-22
fields, referring to fixed-length 2-19
FILES 5-17, 5-202
FINDAP–Find Application Program 5-133
FINISH 2-2
FINISH–End the Application Program 5-137
    END 5-137
FINISH, using 3-1, 3-3
fixed-length fields, addressing 2-19
flag, program check routine (SMSPCR) 2-27
formatting input data 3-5
fullword 4-5

G

Global Machine Segment (GMS) 2-7
global storage 2-12

H

halfword 4-5
hard-copy trace G-1
header, segment 2-13
headers, programming segment 2-19
hexadecimal 4-5
high-resolution counter (HRC) 3-18, 5-195
Host Support 1-3, 2-1, 2-3, 2-5, 3-2, 4-6, 5-67, 5-305, 5-343
  APOPT 2-3
Host Support User's Guide 5-339
  COPY 5-340
  DEFCON 5-340
  DEFLD 5-340
  EQUATE 5-340
Host Transmission Facility 5-103

I

ID, definition of segment space 2-12
idle logical work station, definition of 2-23
immdata 4-5
Index Register Number Table (IRNT) 4-2, F-5, F-10
  BRANX F-9
  LDRA F-9
indexing F-9
  MVDI F-9
  MVFXD F-9
initializing storage 2-12
INITSEG–Initialize Segments 5-139
  ENDINIT 5-139
  SINIT 5-139
INITSEG, using 2-12
INOR–Inclusive OR 5-141
INOR, using 3-12
INORI–Inclusive or Immediate 5-143
INORI, using 3-12
installation diskette 1-3
instruction counter 2-24
instruction step G-1
instructions, arithmetic/logical 3-10
instructions, data operation 3-5
interrupt handling conventions 5-21
interrupt, LPOST 2-25
interrupts, entry point priorities of 2-24
interval timers 5-145
INTMR 3-18
INTMR–Interval Timer 5-145
INTRTBL, defining delimiters with 2-22
IRETURN 3-17
IRETURN–Indexed Conditional Return 5-151
  ADRLST 5-151

J

JUMP 3-14, 4-6
JUMP–Short Branch 5-153
  BRAN 5-153

L

label 4-4
LCF (Local Configuration Facility) 1-3
LCHAP–Change Priority 5-155
LCHAP, controlling dispatching with 2-23
LCONVERT, using 2-29
LDDI–Load Data Immediate 5-161
LDDI, using 3-11
LDFLD–Load Field 5-163
LDFLD, using 3-11
LDFLDC and LDSEGC, using 3-11
LDFLDC–Load Field Character 5-165
LDFP 2-16
LDFP–Load Primary Field Pointer 5-169
LDLN 2-16
LDLN–Load Field Length Indicator 5-171
LDRA–Load Register Address 5-173
LDRA, using 2-17, 3-14
LDREG–Load Register 5-175
LDREG, using 3-11
LDSECT 2-6
LDSECT–DSECT Definition (BEGIN) 5-177
  DEFLD 5-177
  DEFRF 5-177
  Ln 5-177
LDSECT, using 2-15
LDSEG–Load Segment 5-179
LDSEG, using 3-11
LDSEGC–Load Segment Character 5-181
LDSEGLN 2-16
LDSEGLN–Load Segment Length 5-183
LDSFP 2-16
LDSFP–Load Secondary Field Pointer 5-185
leading zero suppression 5-107
LEJECT–Eject to a New Page 5-187
LEJECT, using 3-4
len 4-4
LEND–DSECT Definition (End) 5-189
  DEFLD 5-189
  DEFRF 5-189
  Ln 5-189
LEND, using 2-15, 3-5
length 2-13
length field, use in modified register address 2-18
length, field 2-17
length, specifying segment operand 2-14
letters 4-1
level definition instructions 5-178
LEXEC 3-17
LEXEC–Execute 5-191
  APOPT 5-191
  EQUATE 5-192
  LEXIT 5-191
  LSEEK 5-191
  WRTI 5-191
LEXIT 3-17
LEXIT and LWAIT, using 3-14
LEXIT–End of Processing 5-193
LHRT 3-18
LHRT–Load High-Resolution Counter 5-195
LIFOFF–If Off Then Branch 5-197
LIFOFF, using 3-13
LIFON–If On Then Branch 5-199
  EQUATE 5-199
LIFON, using 3-13
limit of dispatch tables 2-23
limits, 3600 CPGEN usage 2-29
link interrupts, entry point priority of 2-24
link-edit 2-5

link-edited 2-1
link-editing 5-343
link-editing program sections 3-3
linkage edited 2-3
LINKAPB 5-23
LLOAD 2-4
LLOAD–Load an Overlay Section into Main Storage 5-201
    OVLYSEC 5-202
    STOVLY 5-202
LMERGE–Merge Blocks of Records 5-205
Ln–Level Definition for DSECTS 5-213
    LDSECT 5-213
Ln, using 2-15, 3-5
load image 3-2
load point 2-5
log messages, content of system-written 2-27
logical operations 1-5, 3-12
logical work station 1-3, 1-4
logical work station, dispatching the 2-23
logical work station, sharing a 2-12
logical work stations 1-4
loop control 5-40
loop instruction count 2-27
LPOST interrupt, processing an 2-25
LPOST–Post Work Station 5-215
    LEXIT 5-215
    LWAIT 5-215
LREAD–Read Station-to-Station Message 5-217
    LREAD 5-217
LRETURN 2-4, 3-16
LRETURN–Return after a Branch-and-Link 5-219
    BRANL 5-219
    BRANLR 5-219
    LSEEKP 5-219
LSEEK 3-17
LSEEK and LSEEKP, using 3-4, 3-7
LSEEK–Seek (Table Lookup) 5-221
    APOPT 5-222
    DEFCON 5-221
    LSEEK 5-225
    LSEEKP–Extended Seek 5-225
    TABLE 5-221, 5-225
LSEEKPL–Extended LSEEK Parameter List 5-229
LSETOFF 3-15
LSETOFF–Set Off 5-233
    EQUATE 5-233
LSETOFF, using 3-13
LSETON 3-15
LSETON–Set On 5-235
    EQUATE 5-235
    INORI 5-235
LSETON, using 3-13
LSORT–Sort a Block of Records 5-237
    LSORT 5-237
LSPACE–Space a Line of Output 5-241
LSPACE, using 3-4
LTIME 3-17
LTIME–Time (Fixed Format) 5-243
LTIMET 3-17
LTIMET–Time Table 5-247
    LTIMEV 5-247
LTIMEV 3-17
LTIMEV–Time (Variable Format) 5-249
    LTIMET 5-249
LTRT–Translate Input Data 5-253
LTRT, using 3-6
LTRTBEG–Translate Table Begin 5-265
    LTRT 5-265
    LTRTENT 5-265
LTRTBEG, LTRTENT, and LTRTGEN
 using 3-4
LTRTENT–Translate Table Entry 5-267
    LTRT 5-267
    LTRTBEG 5-267
LTRTGEN–Translate Table Generation 5-269
    LTRTBEG 5-269
    LTRTENT 5-269
LWAIT–Wait 5-271
    BEGIN 5-271
    LEXIT 5-272
LWAIT, ending an active session using 2-26
LWRITE–WRITE Station-to-Station Message 5-273
    LREAD 5-273

M

macro library 2-5
main storage 1-4, 2-7
MASK–Mask (For EDIT Instruction) 5-275
    EDIT 5-275
    EQUATE 5-275
MASK, using 3-4
master characters for data compaction 3-8
message fields, defining and processing 2-22
mnemonics 4-8
MOD–Modulus Factor (For MODCHK Instruction) 5-277
    MODCHK 5-277
MOD, using 3-4
MODCHK–Modulus Check 5-279
    APOPT 5-279
    MOD 5-279
MODCHK, using 3-4, 3-6
modified register addressing 2-13, 3-5, 5-89
modified register addressing, description and example 2-17
modules, selecting and loading optional 2-28
modulus 4-7
moving data 3-5
MPYFLD and MPYREG, using 3-11
MPYFLD–Multiply Field 5-281
MPYFLDL–Multiply Field Logical 5-283
MPYREG–Multiply Register 5-285
MPYZ–Multiply Zoned Decimal 5-287
MPYZ, using 3-12
MVCZ–Move and Convert Zoned Decimal 5-289
MVCZ, using 3-5
MVDI–Move Data Immediate 5-291
MVDI, using 3-5
MVFLD–Move Field 5-293
MVFLD, using 3-5
MVFLDR–Move Field Reverse 5-295
    OPTMOD 5-295
    P60 5-295
MVFLDR, using 3-5
MVFXD 2-14
MVFXD–Move Fixed 5-297
MVFXD, using 3-5
MVFXDR–Move Fixed Reverse 5-299
    OPTMOD 5-299
    P60 5-299
MVFXDR, using 3-5
MVSEG–Move Segment 5-301
MVSEG, using 3-5
MVSEGR–Move Segment Reverse 5-303
    OPTMOD 5-303
    P60 5-303
MVSEGR, using 3-5
M45 5-133

N

name 5-27
Nested Overlay Sections 2-5
next sequential instruction 2-25
next sequential instruction (NSI) 5-33
    BRANLR–Branch and Link Register 5-35
    BRANR 5-35
    LRETURN 5-35
non-resident 2-4, 5-20
nonrelocatable 2-2
Nonrelocatable Overlays 2-5
nonrelocatable programs 2-1, 2-2
nsi 5-192

O

operand 1 4-2
operand 2 4-2
operands 4-1
operands, addressing 2-13
operating diskette 1-3
operations, logical 3-12
optional instruction 5-9
optional instruction modules 2-28
optional instruction modules, determining presence of 2-28
options 4-1
OPTMOD 5-9, 5-57, 5-61, 5-65, 5-69, 5-71, 5-75, 5-101, 5-133, 5-145, 5-159, 5-205, 5-225, 5-237, 5-249, 5-253, 5-287, 5-289, 5-322, 5-327, 5-395, F-10, F-13, F-15
    DECOMPTB 5-71
    LTRTBEG 5-255
overflow 4-8
overlaid program sections, defining 3-3
overlay programming 2-1, 2-4
OVLYSEC 2-2, 2-4
OVLYSEC–Define Load Address and Entry Point 5-305
    APOPT 5-305
OVLYSEC-ENDOVLY pair 2-2
OVLYSEC, using 3-3

P

packing/unpacking data with PAKFLD/PAKSEG and UPKFLD/UPKSEG 3-7
pairs, compaction character 3-8
PAKFLD–Pack Field 5-307
PAKFLD, using 3-7
PAKSEG–Pack Segment 5-309
PAKSEG, using 3-7
parameter list 5-55
parameter lists, system-defined DEFxxx 2-26
passing data between programs 3-14
PAUSE–Suspend Processing 5-311
PAUSE, ending an active session using 2-25
PAUSE, using 3-14
PC operand, defining a program check entry point with 2-26
PFP and FLI, source of values for 2-19
PLPCMD–Post-List Processor Commands 5-313
PLPCMD, using 3-4
pointers 2-21
post-list processing 5-313
PRIDSP 5-155
    LCHECK 5-157
    LCHECK ST 5-157
    LCONVERT 5-159
PRIDSP, defining dispatch tables using 2-23
primary and secondary application programs 2-7
primary and secondary programs, defining storage for 2-11
primary application program 2-2, 2-4
primary field pointer (PFP) 2-15
primary-secondary cross-addressing 2-13
PRINTI–Print Macro Expansion 5-315
PRINTI, using 3-4
priority dispatch table 2-23
priority dispatching, work station 2-23
priority, entry point 2-24
private storage 1-4
processing session, definition 2-24
program calling, instructions for 3-14
program check address 2-27
program check codes 2-27, D-1
program checks 1-5
program checks, entry point-related 3-2
program checks, primary/secondary ap processing of 2-28
program control 1-5
program control instructions 3-14
program interrupts, entry point priority of 2-24
program-to-program data transfer 3-14
programmable input facility G-1
publications, related X-1
punctuation 4-1
P2A 5-327
P2C 5-145
P21 5-253
P24 5-225
P26 5-71, 5-75
P27 5-57, 5-61
P31 5-9, 5-65, 5-101, 5-287, 5-289, 5-395
P32 5-249
P34 5-69, 5-159
P5C 5-205, 5-237
P68 5-322, F-10, F-13, F-15

Q

quotation mark 4-5

R

REBASE–Restore the Base Register for a DSECT 5-317
    LDSECT 5-317
    SAVEBASE 5-317
REBASE, using 3-3
reg 2-16, 4-4
register address 5-89
    DEFCON 5-89
    DEFLD 5-89
    EQUATE 5-89
register addresses 4-5
register addressing 2-13
register addressing, description and example 2-16
register 1 2-29
register, using a base addressing 3-3
registers 2-7
registers, allocating extra 2-12
registers, passing 2-29
related publications X-1
release control 2-12
release 3 changes/additions vii
releasing control, instructions for 3-14
releasing work station control 2-25
reloc 2-2
relocatable 2-5

Relocatable Overlays 2-5
relocatable programs 2-1, 2-3
resident 2-2, 2-4
restart 2-24
resuming processing, entry point priority of 2-24
return-address stack 3-16, 5-151
root 2-2

S

SAVEBASE–Save the Base Register for a DSECT 5-319
  LDSECT 5-319
  USEBASE 5-319
SAVEBASE, using 3-3
SCALE–Scale Number 5-321
SCALE, using 3-5
scb (string control characters) 3-8
scratch-pad 3-17
SCRPAD–Scratch Pad 5-327
secondary application program 2-4
secondary application programs 2-2
secondary field pointer (SFP) 2-15
SECTION 2-6
SECTION–Section Control 5-339
  BEGIN 5-339
  OVLYSEC 5-339
  SEGCODE 5-339
SECTION, using 3-3
sections 2-1
sections, defining program 3-3
seg 4-4
SEGALLOC 3-17
SEGALLOC–Segment Allocate 5-341
  APRETURN 5-341
  SEGFREE 5-341
SEGCODE 2-2
SEGCODE–Application Program Section Identifier 5-343
SEGCODE-ENDSEG pair 2-2
SEGCODE, ENDSEG 2-5
SEGCODE, using 3-1, 3-3
SEGCOPY–Segment Copy 5-345
SEGFREE 3-17
SEGFREE–Segment Free 5-349
  APCALL 5-349
  APRETURN 5-349
  SEGALLOC 5-349
segment 1-5, 2-7
segment header 2-13
segment header addressing 2-13, 2-15
segment headers, programming notes about 2-19
Segment Indexing 4-2, 5-20, F-4
segment length indicator (SLI) 2-15
segment number, use in register addressing 2-17
segment space ID, definition 2-12
segment space ID, use in modified register address 2-17
segment space ID, use in register addressing 2-17
segment 0 2-7, 5-19, 5-93, 5-94, 5-139, 5-167, 5-351
  APCALL/APRETURN 5-19
  DEFCON 5-93
  DEFLD 5-93
segment 0 sharing 2-12
segment 0, addressing registers in 2-13
segment 1 2-7, 5-19, 5-79, 5-179, 5-183, B-21
  BEGIN 5-79
  EQUATE 5-79
segment 13 1-5
segment 14 1-5, 5-77, 5-305, B-3
  APOPT 5-77
  LLOAD 5-306
  STOVLY 5-306
Segment 14
  5-9
  DEFCON 5-9
segment 15 1-5, 5-19, 5-83, B-10
segment-displacement addressing 2-13
segment, specifying the 2-13
segments 1-5
segments 2 - 12 2-7
SELECT–Select Segment 0 5-351
SELECT, using 2-12
SETFLDI–Set Field Immediate 5-353
SETFPL 2-16
SETFPL–Set Primary Field Pointer and Field Length Indicator 5-355
  BEGIN 5-355
  DEFCON 5-356
  DEFDEL 5-355
  DEFLD 5-356
SETFPL, example and use 2-19
SETSFP 2-16
SETSFP–Set Secondary Field Pointer 5-361
  BEGIN 5-361
  DEFCON 5-361
  DEFDEL 5-361
  DEFLD 5-361
setting bits on and off 3-13
SETX–Enable/Disable Segment Indexing F-10
SETXREG–Set Index Register Number F-13
Shared Overlay Sections 2-5
shared storage 1-5
sharing storage segments with COBOL programs 2-29
shifting data 3-13
SHIFTL–Shift-Left Data in a Register 5-365
SHIFTL, using 3-13
SHIFTR–Shift-Right Data in a Register 5-367
SHIFTR, using 3-13
shutdown 2-24
SINIT–Start Initialization Section 5-369
  FINISH 5-369
  INITSEG 5-369
SINIT, using 2-12
sms, recording program checks in 2-27
SMSIML 2-22
SMSIML, message definition by 2-22
SMSLSE 3-16
SMSLTC 5-193
  LREAD NOWAIT 5-193
SMSPCR program check routine flag 2-27
SMSUIC 5-193
SNA-Primary 5-29
  DEFLD 5-29
  EQUATE 5-29
SPA 3-17
SPLIT 5-23
  DEFCON 5-24
  OVLYSEC 5-24
  SEGCODE 5-24
split programs F-1
standard definitions 2-26
STARTGEN 5-93
STARTUP operand (STATION CPGEN macro) 2-24
startup, entry point priority of 2-24
STATION 5-27, 5-35, 5-80, 5-94, 5-147
  APCALL 5-28
  APOPT 5-27
  BEGIN 5-94
  BRANL 5-35
  DEFDEL 5-27
  DEFSTOR 5-28

LEXIT 5-28
LPOST 5-28
LSEEKP 5-35
SETFPL 5-80
SETSFP 5-80
station chain dispatching 2-23
statistical counters 5-119
STATS–Obtain or Reset Extended Statistical Counters 5-371
status 4-8
status codes 1-5
STFLD–Store Field 5-379
STFLD, using 3-11
STFLDC 3-11
STFLDC–Store Field Character 5-381
storage allocation 2-7
storage definition 2-7, 2-12
storage definition, primary/secondary program 2-11
storage initialization with SINIT, INITSEG, and ENDINIT 2-12
storage pools 1-4
storage segments 1-5
storage, COBOL sharing 2-29
storage, defining a dump work area in 3-4
storage, initializing 2-12
STOVLY 2-4
STOVLY–Start Overlay 5-383
APOPT 5-383
ENTRY 5-383
EXTRN 5-383
OVLYSEC 5-383
STOVLY 5-383
STP (startup) operand of BEGIN 2-24
string control characters (SCB) 3-8
STSEG–Store Segment 5-385
STSEG, using 3-11
STSEGC 3-11
STSEGC–Store Segment Character 5-387
SUBFLD and SUBREG, using 3-11
SUBFLD–Subtract Field 5-389
SUBFLDL–Subtract Field Logical 5-391
SUBREG–Subtract Register 5-393
subroutine programming 2-1, 2-4
subsystem library 5-67
Subsystem Operating Procedures 1-3
SUBZ–Subtract Zoned Decimal 5-395
SUBZ, using 3-12
syntax rules 4-1
SYSAP 5-103
DEFCON 5-103
system configuration 1-4
system log, program check messages in 2-27
System Machine Segment (SMS) 2-7
system monitor 1-3, 5-103, 5-345, G-1
DTAFREE 5-103
Systems Network Architecture/Synchronous Data Link Control (SNA/SDLC) 1-2

T

table definition with DEFCON 3-7
table location, delimiter 3-2
table lookup using LSEEK and LSEEKP 3-7
TABLE–Define Table for LSEEK/LSEEKP 5-397
APOPT 5-398
LSEEK 5-397
LSEEKP 5-397
TABLE, using 3-4, 3-7
tables 2-7
tables, dispatch 2-23
terminal interrupts, entry point priority of 2-24
testing bits 3-13
TESTX–Test for Active Indexing F-15
TF1,TF2, TF3, TF4 5-17
Threshold Analysis and Remote Access (TARA) 5-146
time-of-day 5-243
time-of-day timer 3-17
time-stamp 3-18
timer 2-25
timer interrupts, entry point priority of 2-24
TRANPL 5-19
transient 2-2, 2-4
translating data with LTRT 3-6
translation table 3-5
TSTMSK–Test under Mask 5-399
TSTMSK, using 3-13
TSTMSKI–Test under Mask Immediate 5-401
TSTMSKI, using 3-13
twos complement 5-165
LDFLDL 5-167

U

underscoring 4-1
unpacking data 3-7
UPKFLD–Unpack Field 5-403
UPKFLD, using 3-7
UPKSEG–Unpack Segment 5-405
UPKSEG, using 3-7
USEBASE–Use a Base Register for a DSECT 5-407
LDSECT 5-407
USEBASE, using 3-3
user-written macros 5-315

V

variable-length fields, defining 2-20
variables 4-3
VERIFY–Verify 5-409
VERIFY, using 3-6
verifying data 3-6
version number, specifying a program 3-2
VIEW–VIEW APCALL/APRETURN Stack 5-411
APCALL 5-411
APRETURN 5-411
Volume 2: Disk and Diskette Programming iii
Volume 3: Communication Programming iii
Volume 4: Loop and Device Cluster Adapter iv
Volume 5: Cryptographic Programming iv
Volume 6: Control Program Generation iv

W

work station 2-7
work station dispatching 2-23
WRTI 3-17

Y

YL2 and AL2 address constants, specifying 3-1

Z

zoned decimal 3-11
zoned decimal arithmetic 3-10

3

3600 Finance Communication System F-1
    APOPT F-1
    BEGIN F-1
    DEFCON F-1
    LEXEC F-1
    LLOAD F-2
    LSEEK F-2
    LSEEKP F-2
    LSEEKPL F-2
    OVLYSEC F-3
    SECTION F-3
    SEGCODE F-3
    SEGCOPY F-3
    STOVLY F-4
    TABLE F-4
3600 programs, using 2-29

4

4331 Multiuse Communication Loop protocol 1-2
4700 Terminals 1-2

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

4700 Finance Communication System
Controller Programming Library
Volume 1
General Controller Programming

Order No. GC31-2066-1

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate. Comments may be written in your own language; English is not required.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*
Possible topics for comment are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address, and date:

____

____

____

____

What is your occupation?______________________________

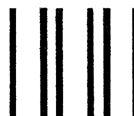
Number of latest Newsletter associated with this publication:____________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

GC31-2066-1

Reader's Comment Form

Fold and Tape | Please Do Not Staple | Fold and Tape



NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 78C
1001 W.T. Harris Boulevard
Charlotte, NC, USA 28257

Fold and Tape | Please Do Not Staple | Fold and Tape

4700 Finance Communication System Controller Programming Library Volume 1
General Controller Programming (File No. S370/4300/8100/S34-30) Printed in U.S.A. GC31-2066-1

Publication Number
GC31-2066-1

File Number
S370/4300/8100/S34-30

Printed in
USA

IBM