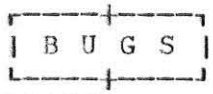


BOGUS

Brown University Graphics System[1]

LEVEL0 Extended Machine

Principles of Operation

The Brown University Graphics Project

Division of Applied Mathematics

Box F

Brown University

Providence, Rhode Island 02912

Updated: June 25, 1974

Printed: September 15, 1976

---

[1]This research is being supported by the National Science Foundation Grant GJ-29401X, the Office of Naval Research, Contract N00014-67-A-0191-0023, and the Brown University Division of Applied Mathematics; Principal Investigator Andries van Dam.

TABLE OF CONTENTS

Abstract

This publication describes the LEVEL0 Extended Machine, a component of the Brown Operating Graphics University System, running on the BUGS META 4A processor. LEVEL0 provides the BUGS user with facilities beyond those inherent in the hardware/firmware. A thorough knowledge of the Principles of Operation of the META 4A is assumed.

1 INTRODUCTION

LEVEL0 is a software package designed to run on the META 4A and provide the BUGS user with facilities making up an extended machine. These facilities alleviate some of the drudgery of coding on a bare machine and assume responsibility for many machine functions, freeing the user to worry about the operation of BUGS (referred to as the "system") at a higher level. LEVEL0 assumes the presence of another level of operating system, running as a "LEVEL1" above it. Such a LEVEL1 does exist on BUGS, and is called the Graphics Monitor System (GMS), or, to close friends, "Our Fearless Leader". In this publication, however, it will be referred to as LEVEL1 for generality.

The reader is assumed to have a thorough knowledge of the META 4A and BUGS in general. In particular, however, he will use only those "normal" instructions not concerned with machine status, interrupt handling, I/O, etc. LEVEL0 is responsible for handling those portions of the machine. Thus it is necessary for the reader to realize that much of what he reads in the META 4A Principles of Operation will not directly concern him.

2 THE LEVEL1 ROUTINE AND ITS STACK FRAME

2.1 LEVEL1 VS. THE BARE MACHINE

A LEVEL1 routine running on LEVEL0 must follow somewhat different conventions from one running stand-alone on the META 4A. These conventions are necessary for two reasons. First of all, they allow LEVEL0 to maintain control over the status of the system, and secondly, they provide a higher level of system organization for the LEVEL1 programmer.

2.1.1 NON-PRIVILEGED STATE

The LEVEL1 routine runs at all times with the Privilege bit in the MSR off. This prevents him from performing local I/O and S/360 operations without using the facilities provided by LEVEL0. Specifically, he cannot execute the following instructions: SIOR, SIO, RST, WST, EXCC, TRB, and SS. It is considered invalid for the LEVEL1 system ever to set the Privilege bit on.

2.1.2 WAIT STATE

In addition to the Privilege bit, the IEVEL1 program must never set the Wait bit in the MSR. LEVEL0 assumes all control of the state of the system, including the setting of this bit.

2.1.3 EXTENDED INSTRUCTIONS

Because of the abcve restrictions and for the increased flexibility of LEVEL1, LEVEL0 provides a set of "Extended Machine Instructions". These instructions, which are coded by the programmer as if they were real META 4A instructions, provided added features for LEVEL1, such as memory management,extended I/O, etc. These instructions, of

which there are some fifteen, will be explained in the course of this publication.

2.2 THE PROGRAMMER'S DATA

Since computing is concerned with the manipulation of data, strict conventions concerning types of data are made by LEVEL0. There are specifically three types of data with which the programmer can concern himself: static, automatic, and controlled.

2.2.1 STATIC DATA

Static data is data that is assembled/compiled into a routine and is present there when the routine is loaded. This data includes the instructions of the program itself, plus any pre-initialized variables DCed within the routine proper. The term "static" may be misleading in that it is not assumed that static data will remain unchanged throughout the execution of a program; it is certainly possible for a programmer to DC an initial value for a variable and then change it later. The term static simply implies that the data was defined before execution of the routine was begun.

2.2.2 AUTOMATIC DATA

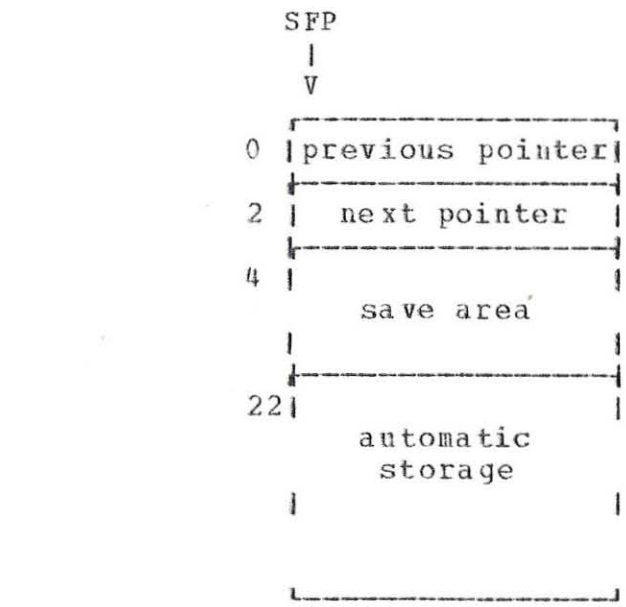
Automatic data is the term applied to variable space which is allocated prior to the execution of a routine and freed when that routine completes. This data cannot be initialized at assembly/compile time, because the storage does not exist at this point. The data is local to the routine owning the automatic storage and disappears when that routine completes. It is this type of data that is specially treated by LEVEL0.

2.2.3 CONTROLLED DATA

Controlled data is data which is maintained in storage space obtained by a routine and present until exclicitly released. Extended instructions are provided to maintain this controlled storage. Most programmers are probably familiar with this type of storage from experience with IBM Operating System/360 or similar systems, with their GETMAIN and FREEMAIN SVCs.

2.3 THE STACK FRAME

As was mentioned above, automatic storage is one special feature of LEVEL0. In order to maintain this automatic storage across normal occurences such as subroutine calling and the execution of interrupt handlers and the like, LEVEL0 maintains a number of "stack frames" in the META 4A. A stack frame is a logically infinite piece of storage in which is maintained a series of automatic storage sections for each routine in the dynamic sequence of execution. Each routine need only be concerned with the section of the stack frame belonging to him. This section has the following format:



previous pointer: this halfword contains the address of the stack frame section of the routine executing dynamically prior to this routine.

next pointer: this pointer is used by LEVEL0 to maintain the dynamic link of stack frame sections and is of no direct use to the programmer.

save area: these 15 halfwords are used to store the routine's MSR through register 14 at any time his execution is delayed due to an actual machine interrupt or to a subroutine call.

automatic storage: This space is the actual automatic storage requested by the routine. It can vary in size from 1 to n halfwords; this size is determined by the extended instruction ENT, which must be the first instruction of every routine that either saves the registers of the previous routine or requires automatic storage or both.

2.4 SUBROUTINE LINKAGE INSTRUCTIONS

ENTer routine ENT RI

The immediate halfword of this RI-format extended instruction specifies the size, in bytes, of the automatic storage required by the routine. The size is rounded up to the next higher halfword, if necessary, and used to determine the size of the automatic storage shown in the diagram above. This instruction must be the first executed instruction of every routine in which it appears.

Once the ENT instruction is performed, the caller's registers 0 through 14 are saved in the caller's savearea, and the Stack Frame Pointer (SFP), register 15, is set to point at the stack frame section for the ENTered routine. This register may be used as a base for instructions accessing data out of the stack frame, but should not be modified in any way by the routine. Its contents are maintained entirely by LEVEL0.

Once a routine has completed execution, it must return control to the routine dynamically previous to it. This is done by executing the RET extended instruction:

RETurn from routine RET RR

This instruction, which has no operands, causes the stack frame section allocated to the routine to be freed up, and control to be returned to the interrupted point of the previous routine. The SFP will be backed up so as to be

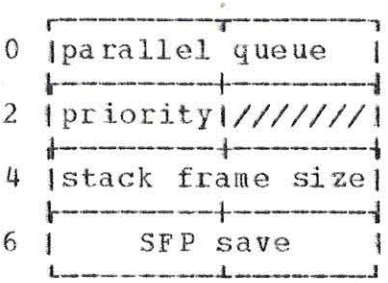
correct for the prior routine and registers 0 through 14 restored from its savearea. If the returning routine desires to modify the registers of the previous routine, an action which is entirely valid in certain cases, he may do so by picking up the previous pointer and using it as a base to address the previous registers. If the returning routine is re-entered, its automatic storage will have been in no way preserved. The contents of automatic storage are always undefined at the start of a routine.

2.5 TYPES OF ROUTINES

At this point in time we can begin to make a distinction between various types of routines, namely parallel routines, subroutines, and immediate routines.

2.5.1 PARALLEL ROUTINES (tasks)

A parallel routine is one which can run simultaneously with and independently of any and all other routines. LEVEL0 has the capability to run an arbitrary number of parallel routines simultaneously within the system. Each parallel routine is given control and allowed to execute until some event occurs that LEVEL0 decides should cause another parallel routine to gain control. Each parallel routine has its own "infinite" stack frame which remains in existence until that routine returns. At the front of that stack frame is a section of storage known as the "stack frame header", which has the following format:



parallel queue: this pointer is used by LEVEL0 to maintain a queue of the stack frames of each of the parallel routines currently running on the system. The head of this queue is in memory location X'60'.

priority: This byte contains the priority assigned to this parallel event. The priority is used by LEVEL0 to decide which parallel event should be given control each time such a decision must be made. See Section 3.2.

stack frame size: This halfword contains the stack frame size estimation made by the programmer. See Section 3.1.

SFP save: Whenever the parallel routine is not executing, its current SFP is saved in this halfword.

The previous pointer in the stack frame section of a parallel event is zero because there is no dynamically previous routine. Each parallel event is considered an entity in itself and control is never passed between them under programmer request.

Also associated with the entry into such a routine is some pre-determined data called "status". This data is stored into the first few bytes of the routine's automatic storage by LEVEL0 before that routine is placed on the queue. The programmer must include this space in his automatic storage request in the ENT instruction. This data will be explained in greater detail later.

2.5.2 SUBROUTINES

A subroutine is a routine which is explicitly invoked by another routine. Although the subroutine is logically a separate routine with its own automatic storage, LEVEL0 considers it to be an extension of the invoking routine. It gets its stack frame section out of the stack frame of the invoking routine, and it runs with the priority of the originally entered parallel routine.

The programmer exercises explicit control over the execution of subroutines; in order to invoke one he must load the return address into register 14 and branch to the ENT instruction of the subroutine, via:

```
BAL     R14,subroutine
```

or, if the subroutine is external:

```
LI      Rx,V(subroutine)
BALR    R14,Rx
```

When the subroutine is entered, the contents of the registers will be identical to what they were in the invoking routine, except of course for the SFP. It is not necessary for the subroutine to restore any registers before exiting with a RET instruction; this is taken care of by LEVEL0 with the save areas.

A subroutine itself may call other subroutines, down to any level. If a subroutine desires to modify the registers of the invoker, he may do so as explained in Section 2.3, by using the previous pointer as a base for the invoker's stack frame section. A typical case of this is for implementing return codes.

2.5.3 IMMEDIATE ROUTINES

An immediate routine, unlike a parallel one, cannot run simultaneously with other routines. The purpose of the immediate routine is to perform system maintainence functions, such as the handling of I/O interrupts, as quickly and with as little overhead as possible. To accomplish this, a immediate routine must:

1) Run disabled, i.e., with the I/O and S/360 interrupt masks in the MSR off. This ensures that the execution of the routine will not be interrupted by any external unit.

2) Not issue any extended instructions which would cause another parallel routine to gain control of the system. These restrictions will be described under the appropriate instructions.

An immediate routine, however, may call subroutines and may cause other immediate routines to gain control. Any subroutine called must also follow the above conventions. An immediate routine and all its subroutines, because they do not run in parallel with other routines, do not have their own stack frame. Instead, they run in the stack frame of the parallel routine that was executing at the time the immediate routine was invoked.

2.5.4 REGISTERS AT ENTRY TO A ROUTINE

The following table describes the status of the registers upon entry to a parallel or immediate routine:

MSR: Condition Code and Flag zero; Arithmetic Overflow and Stack Overflow/Underflow disabled; interrupt masks as described above. Parity interrupts are always enabled, and the Parity interrupt mask bit in the MSR should not be altered.

PC: Set appropriately.

R2 - R3: Identical to what they were when the formerly executing routine was interrupted. This is especially useful for SVC handling routines, which are passed parameters in these registers.

R4 - R14: Undefined.

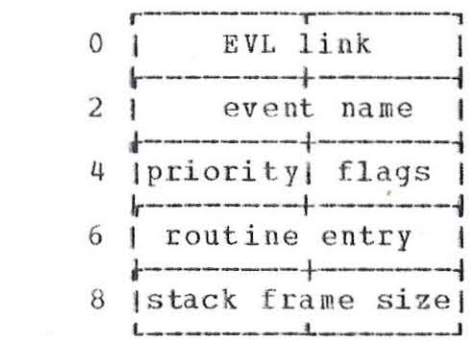
SFP: Set appropriately.

3 EVENTS AND THEIR ROUTINES

Now that the various types of routines available on LEVEL0 have been discussed, it is time to describe how individual routines are invoked. The simplest case is that of subroutines, which are explicitly invoked by the programmer as described in Section 2.5.2. It is parallel and immediate routines that this section deals with.

3.1 EVENTS

An event in BUGS is defined as the occurence of some sort of system interrupt that should delay the execution of the current routine and start up another. Events are such things as I/O interrupts, SVC calls, Program Checks, etc. LEVEL0 presumes that the LEVEL1 system will have a routine which should be executed when each of these events occurs, and therefore must invoke the proper routine at the proper time. In order to do this, LEVEL0 needs an EVent List (EVL), which is a list of entries, each one specifying an event and the routine to be invoked when it occurs. Each event entry has the following format:



EVL link: This is the address of the next entry in the EVL.

event name: This is the 16-bit name assigned to the event. Many events have names pre-assigned by LEVEL0 so that an effective communication with LEVEL1 can be set up. Other event names can be assigned by LEVEL1. The first hex digit of the name is called the event "type", and is used in searching the EVL.

priority: This is the 8-bit priority assigned to the event routine if it is parallel. It is used in determining which parallel routine to run, as described below.

flags: This byte contains flags describing the event routine:

bit 0: If zero, this routine is immediate. If one, the routine is parallel. The priority is ignored for immediate routines.

bit 1: If on, the event is ignored when it occurs and no routine is invoked. This bit may be altered dynamically by the LEVEL1 system.

routine entry: This is the address of the routine itself. The first instruction must be an ENT specifying the amount of automatic storage desired by the programmer.

stack frame size: This halfword is only used for parallel routines. As was mentioned previously, the stack frame is a logically infinite piece of storage. In actuality, it is composed of one or more stack frame extensions. This halfword should contain an estimation of the total amount of stack frame space needed, so that the first extension will hopefully be the only one. This cuts down on LEVEL0 overhead and the fragmentation of memory.

In order to locate the event entry for an event when it occurs, LEVEL0 uses a table of EVL "heads"[2]. There are sixteen halfword event heads, located in memory locations 70 through 8E. Each head is used to point to the EVL for the corresponding event type. It is up to the LEVEL1 system to set up the event head table in its initialization code. The sixteen event types are:

TYPE	DESCRIPTION
0	Interval Timer events
1	META 4B events
2	All other local I/O unit events
3	S/360 events
4	)
5	)
6	)
7	> events for LEVEL1 use
8	)
9	)
A	)

---

[2] *vide* Section 8 for the location of these lists.

B)
C SVC events
D Program Interrupt events
E Program Manipulation events
F Event trap

The following steps are taken when an event occurs:

1) LEVEL0 builds the appropriate event name and extracts the first hex digit as the event type.

2) The event type is used to pick an EVL head from the above table, and the EVL is searched for an entry containing the name.

3) If none is found, the fourth hex digit of the event name is zeroed and the list is searched again. This allows generic classes of events.

4 and 5) If still no event entry is found, two more searches are made, with the third and second hex digits, respectively, also zeroed.

6) Finally, if the above searches were unsuccessful, a check is made of event type F, the event trap, to see if any entry exists in that EVL, regardless of name.

It is presumed that one of the above searches produces an event entry which can be used to invoke a routine. If no entry was produced, the event is ignored and discarded. If an entry was found, the follcwing actions occur:

parallel event: A new stack frame is created with the size specified in the entry. The header is initialized with the priority and the frame size and placed at the head of the queue of stack frames.

immediate event: The routine is entered after allocating it space in the current stack frame in accordance with its ENT instruction.

No action is taken of course, if the ignore flag is on in the event entry.

3.2 DISPATCHING OF PARALLEL ROUTINES

There are certain conditions under which LEVEL0 determines that the current parallel routine should be delayed and another one given control. Whenever this occurs, the LEVEL0 "dispatcher" is executed. It performs the following search:

1) A scan of the parallel queue is made for the highest priority runnable parallel routine. A routine is not runnable if it has gone into wait state (see Section 3.3.1). If two or more have the highest priority, the last one on the queue is picked.

2) If all routines are in wait state, the first one is picked.

Once a routine is picked, it is given control by loading its current SFP from the stack frame header and picking up its registers from the save area. During its execution the address of the predecessor stack frame header on the event queue will be in memory location X'62'.

3.3 LEVEL1 CONTROL OF ROUTINES AND EVENTS

Certain extended instructions are provided by LEVEL0 to assist LEVEL1 in controlling the execution of routines and the occurence of events.

3.3.1 WAIT AND POST

It is possible for a routine to put itself into wait state, i.e. a state where execution is suspended, pending notification or "posting" by another routine. Wait state is controlled by the Wait Control Halfword (WCH), used as the communication link between the waiter and the poster.

WAIT on wch WAIT RX

The second operand is a WCH, which is checked to see if bit 0 is on. If not, the SFP of the routine issuing the WAIT is set into the WCH, and the routine is flagged as being in wait state by setting the Wait bit in its MSR. If bit 0 is

on, the WCH has already been posted and the routine is allowed to continue. An immediate routine may not issue a WAIT.

POST wch POST FSS

The second operand is a WCH, which is posted by turning off the Wait bit in the MSR of the routine whose save area it points to. The first operand address is then moved into the WCH as a completion code and bit 0 is set on.

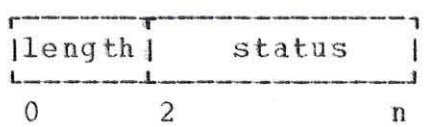
A typical use for these instructions is for waiting for an I/O operation to complete. A routine starts some I/O and then WAITS on a WCH which is POSTed by the I/O interrupt handling routine.

3.3.2 SIGNAL

The SIGNAL extended instruction is provided to allow LEVEL1 to force the invocation of an event routine.

SIGNAL event SIGNAL FSS

The second operand halfword is the name of an event whose occurence is to be forced. The event may be parallel or immediate and its routine will be invoked provided the EVL search is successful and the ignore bit is off. The first operand address points to the status to be placed in the beginning of the routine's automatic storage. This status must be in the following format:



An immediate routine cannot SIGNAL a parallel one.

4 MAINTAINING CONTROLLED DATA

A group of extended instruction are provided so that LEVEL1 can easily maintain controlled storage. Contiguous areas of controlled storage are maintained in a linked list called the Free Memory List (FML). The head of this list is in memory location X'64'.

4.1 GETTING CONTROLLED STORAGE

GET controlled storage Register	GETR	RR
GET controlled storage	GET	RX

The contents of R2 (GETR) or the second operand halfword (GET) specifies the amount of controlled storage desired. This size is rounded up to a multiple of four bytes and the storage is obtained from the FML. Its address is returned in R1. If no contiguous piece of storage of the requested size exists, a No Free Memory Program interrupt occurs.

GET MAXimum controlled storage	GETMAX	RR

The address of the largest contiguous piece of free controlled storage is returned in R1, and its length is returned in R2. If absolutely no storage is available, a No Free Memory Program interrupt occurs.

4.2 FREEING CONTROLLED STORAGE

FREE controlled storage Register	FREER	RR
FREE controlled storage	FREE	RX

The contents of R1 contains the address of an area of controlled storage to be FREEd, i.e., put back on the FML. The contents of R2 (FREER) or the second operand halfword (FREE) contains the length of this area. If the address is not even an Invalid FREE Program interrupt will occur.

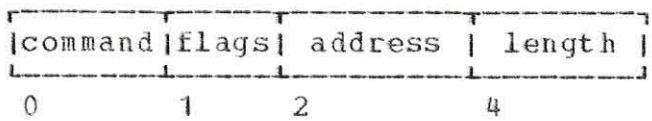
5 EXTENDED I/O

LEVEL0 provides a great many extended I/O facilities to the LEVEL1 programmer, both in terms of local I/O and S/360 communications. These facilities provide a higher level of control over I/O and I/O interrupts, allowing LEVEL1 routines to be much smaller and logically simpler. In addition, they eliminate all of the data codes associated with various units and reduce all data transfer to the EBCDIC code.

5.1 LOCAL I/O UNITS AND THE CHANNEL

5.1.1 THE CHANNEL PROGRAM

LEVEL0 provides a logical "channel" which has the capability of executing multiple operations (IOCCs) at a unit via only one LEVEL1 instruction, EXCP, which initiates a chain of unit commands. These commands are known as Channel Program Commands (CPCs) and have the following format:



command: This byte specifies the particular operation to be performed by this CPC at the unit. There are seven different commands:

x1- Write. Data is transferred from the memory address specified to the unit. The length specifies the number of bytes to transfer.

x2- Read. Data is read from the unit into memory at the address specified. The length determines the number of bytes read.

x3- No Operation. No operation is performed at the unit. The address and length fields are ignored.

x4- Sense. The USH is read into the halfword at the specified address. The length field is ignored.

x5- Sense with Reset. This CPC operates exactly as Sense with the addition that any pending interrupt from the unit is reset.

x6- Special. This CPC command has special usage with different units. See the explanation of each unit.

x7- Transfer in Channel. This CPC is not used to perform I/O at the unit, but rather to cause a branch so that the channel program may be continued at a different memory location. The address field specifies from where the next CPC is to be obtained; the length field is ignored.

flags: The only flag currently used is bit 0. This bit is checked after completion of each CPC except a Transfer in Channel, and, if on, another CPC is retrieved from the next three halfwords in memory. This feature, with Transfer in Channel, allows initiating controllable multiple operations at the unit with only one channel program.

address: This field specifies the memory address where data associated with the command is to be obtained or stored.

length: This field gives a byte count for data transfer commands.

5.1.2 STARTING A CHANNEL PROGRAM

EXecute Channel Program EXCP FSS

The second operand address is the address of the first CPC in the channel program. This program is initiated, if possible, at the local unit specified by the low-order 4 bits of the operand 1 address. The Condition Code is set as follows:

C0- Unit busy or offline.
C1- Channel program in progress.
C2- Channel program completed immediately.

Once a channel program that caused C1 to be set is completed, an event with a pre-defined name is signalled by LEVEL0. This is to inform LEVEL1 of its completion and any error conditions that occured, and to allow LEVEL1 to perform any post-I/O housekeeping necessary. Appendix I lists these event names and their associated status information.

5.1.3 OTHER LOCAL I/O FACILITIES

In order to control I/O interrupts, it is perfectly valid for LEVEL1 to manipulate the I/O mask bit in the MSR and the I/O unit mask in location 2E, within limits. It is considered invalid for a immediate routine to enable I/O interrupts.

In addition, LEVEL1 can never modify the UCB table or the UCBs for any unit. However, the third halfword of each UCB is reserved for LEVEL1 use as a pointer to a UCB extension or whatever. LEVEL1 may access this halfword through the UCB table and modify it at will.

5.1.4 HANDLING OF PARITY CHECKS

When a Parity Check Control Panel interrupt occurs, LEVEL0 goes into "hard" stopped state. The user can only recover by resetting the system and re-IPLing.

5.2 S/360 COMMUNICATION

Although S/360 I/O logically belongs on LEVEL0, it is currently supported by a subroutine package which dynamically "hooks" itself into LEVEL0. See S/360 - META 4A I/O, Subroutine Descriptions for calling conventions.

5.3 THE INTERVAL TIMER

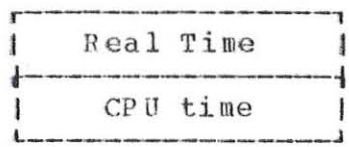
The Interval Timer decrements location X'50' every 100 usec. When location X'50' goes to zero, it causes an Interval Timer interrupt. The LEVEL0 support for the Interval timer allows the LEVEL1 user to keep track of running and CPU time, and to set time intervals.

5.3.1 CPU AND RUNNING TIME

The user may query the running and CPU time by use of the QTIMER instruction:

Query TIMER QTIMER RX

The first and only operand address is the address of an eight byte scannout area, aligned on a halfword boundary. After completion of the QTIMER instruction, the scannout area contains the running time and CPU time since the last IPL in timer units (100 usec.):



5.3.2 TIME INTERVALS

crude!

The LEVEL1 user may set a time interval by placing the real time at which he wishes to be interrupted in the Timer Unit Control Block extension. When that time is reached, LEVEL0 will signal an interrupt to LEVEL1.

5.4 META 4B COMMUNICATIONS

One instruction is provided by LEVEL0 for communications with the META 4B. The META 4A can interrupt the META 4B and pass it a code using the INTB instruction:

INTerrupt meta 4B INTB RX

The first and only operand address is passed to the META 4B to identify the interrupt.

6 UNIT-DEPENDENT CPCS

This section describes the CPCs which are unit-dependent, i.e., Write, Read, and Special. Note that all data transfer is in EBCDIC.

6.1 3461 CARD READER

01- Write. This command is invalid for the 3461 and, if encountered, causes the channel program to be aborted and an Invalid CPC Program interrupt to occur. This is the case for all invalid CPCs for every unit.

02- Read. This command causes the number of bytes specified by the length field to be read into memory starting at the address in the CPC. If the length is not in the range 1 to 80, unpredictable results will occur.

06- Special. Invalid.

6.2 INTERVAL TIMER

01- Write. Invalid.

02- Read. Invalid.

16- Start Timer. The Timer begins to decrement location X'50' every 100 microseconds.

06- Stop Timer.

6.3 4132 KEYBOARD/TYPEWRITER

01 or 11- Write or Write without Edit. The character string specified by the address and length fields is typed on the Typewriter, with trailing blanks removed and a carriage return appended. If an exact typing of the

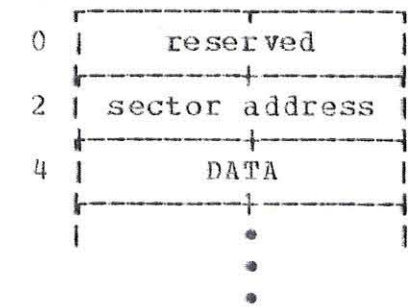
string is desired, with trailing blanks and no carriage return, then bit 3 of the command should be set.

02 or 12- Read or Read without Edit. Input is accepted from the keyboard until either a carriage return is typed or the length is exhausted. If a Read is performed, as opposed to a Read without Edit, then two extra facilities are provided: Logical Backspace and Logical Line Delete. If a logical backspace character is typed (LEVEL1 specifies this character in location 66), the previous character is ignored and removed from the input buffer. If a logical line delete is typed (also specified by LEVEL1 in location 67), the complete preceeding input is ignored and the Read is restarted.

06- Special. Invalid.

6.4 1444 DISK STORAGE UNIT

x1- Write. Data is written onto the disk from the buffer pointed to by the CPC address. This buffer has the following format:



The length is rounded up to a halfword and used to determine the amount of data to write. Any amount may be written with one CPC. The sector address specifies the first sector on which data is to be written; an automatic seek to this sector is performed by LEVEL0. If the length is not a multiple of 640 bytes, the last sector will be filled with zeroes.

02 or 12- Read or Read Check. The Read command uses the same buffer format as a Write, and performs equivalently in terms of length specification, automatic seek, etc. The only difference is that data is read into the buffer and the length does specify the exact amount of data to transfer.

Read Check is used to check data just written on the disk and should be chained onto every Write for best performance. The address and length fields should be identical to those in the Write.

06- Special. This CPC is used to cause a "stand-alone" seek of the carriage arm. This can cause a performance increase in certain situations where computing is to be done before a Read or Write - the seek can operate concurrently with the computing. The address field points to a halfword containing the sector address to which the seek is done; the length field is ignored.

6.5 CONTROL PANEL

x1- Write. Either one or two halfwords at the address specified are written on the lights, depending on the setting of bits 2 and 3:

00: No Operation.
01: One halfword is written on the lowers.
10: One halfword is written on the uppers.
11: First halfword is written on the uppers and the next one on the lowers. The length field is ignored.

x2- Read. The contents of the data switches are read into the halfword at the specified address; the length field is ignored.

x6- Special. Invalid.

6.6 NULL META 4B

01- PIO Write. The halfword at the data address is used to perform a PIO write to the Vector General. The length field is ignored.

02- Read. The length field divided by two specifies the number of consecutive registers to be read from the Vector General into the memory location specified. The starting register address is determined by the last PIO Write CPC.

06 or 16- Allow Interrupts or Set Display Buffer Address. The Allow Interrupts command is used to inform the META 4B that it can again request interrupts. This CPC should be issued at the end of the interrupt handler for the B.

In a Set Display Buffer Address Command, the display buffer address specified in the CPC is sent to the Null META 4B so that it can initialize for displaying.

Both of these CPCs ignore the length field.

7 APPENDIX I: LEVEL0-DEFINED EVENTS

This section lists and describes the events that are pre-defined by LEVEL0. In addition, the status data passed to the event handling routine in its automatic storage is explained.

The first halfword of the status, and therefore of an event's automatic storage, is always the event name that was originally signalled (as opposed to the name actually found in the EVL search). This is also true for events caused by LEVEL1 via the SIGNAL instruction. The remaining status is variable, although two halfwords are common:

Last CPC Address: On an I/O event, this is the address of the last CPC that was interpreted. Some CPCs may be ignored if an error condition arises.

USH: On an I/O event, this is the final USH from the unit causing the event.

7.1 INTERVAL TIMER EVENTS

2001- The time of day placed in the UCB extention by LEVEL1 has been reached. Status is the running time (32 bits) and CPU time (32 bits) at the time of the interrupt, in timer units.

7.2 META 4B EVENTS

1031- META 4B interrupt. Status is the code specified in an INTA instruction executed on the META 4B.

7.3 NULL META 4B EVENTS

1031- Null META 4B interrupt. Status is the USH and the display buffer address. Further interrupts will be enabled after an Allow Interrupts CPC is executed.

7.4 OTHER LOCAL I/O UNIT EVENTS

The second hex digit of the event name for these events is the unit address of the local I/O unit.

7.4.1 3461 CARD READER

2411- Channel Program Complete. Status is Last CPC Address and USH.

2421- I/O Error, caused by Read or Feed Check, or Hopper Empty. Status is the same as 2411.

7.4.2 4132 KEYBOARD/TYPEWRITER

2611- Channel Program Complete. Status is Last CPC Address, USH, and Remaining Length. The Remaining Length is only meaningful on Read commands, where it gives the difference between the length specified in the CPC and the actual number of characters typed in.

2631- Interrupt Switch. Status is USH.

7.4.3 1444 DISK STORAGE UNIT

2211- Channel Program Complete. Status is Last CPC Address and USH.

2221- Seek Check, caused if the sector number on an automatic or stand-alone seek cannot be verified after ten retries. Status same as 2211.

2222- I/O Error, caused by Read or Read Check commands if a data transfer error persists after 10 retries. Status is same as 2211 and 2221.

7.4.4 CONTROL PANEL

2531- Interrupt Button. There is no status other than the event name.

7.5 SVC EVENTS

The event name for an SVC instruction is always C0xx, where xx is the SVC code. No status other than the name itself is passed.

7.6 PROGRAM INTERRUPT EVENTS

The event name for a Program Interrupt is always D0xx, where xx is the Program interrupt code. No status other than the name itself is passed, except in the following cases:

Operation- An image of the Program Interrupt Scan-out Area is passed, comprising five halfwords of status.

Invalid CPC- The Last CPC Address is passed. This CPC contains the invalid command.

7.7 PROGRAM MANIPULATION EVENTS

This event is really LEVEL1-defined, but is intended for use in initially starting up the LEVEL1 system and in multi-programming maintenance. See the appropriate LEVEL1 manual for an explanation.

7.8 EVENT TRAP

As described in Section 3.1, this event is used if no other event entry was found in the EVL search. The first entry in the event trap EVL is used, and should have a standard name of F000.

8 APPENDIX II: LEVEL0 LOWER MEMORY LAYOUT

The following table lists the various lower memory areas in LEVEL0 which are accessable by LEVEL1:

LOCATION	CONTENTS
2E	I/O unit interrupt mask
30	UCB Table - Interval Timer UCB address
32	META 4B UCB address
34	1444 Disk Storage Unit UCB address
36	
38	3461 Card Reader UCB address
3A	Control Panel UCB address
3C	4132 Keyboard/Typewriter UCB address
3E	META 4A UCB address
40	SIMALE UCB address
42	Vector General UCB address
44	
46	S/360 Device 050 UCB address
48	S/360 Device 051 UCB address
4A	S/360 Device 052 UCB address
4C	S/360 Device 053 UCB address
4E	
50	Interval Timer
60	Parallel Queue head
62	Pointer to predecessor of executing routine
64	Free Memory List head
66	Logical Backspace character
67	Logical Line Delete Character
70	Event List heads - Timer events
72	META 4B events
74	all other local I/O unit events
76	S/360 events
78	)
7A	)
7C	)
7E	> events for LEVEL1 use
80	)
82	)
84	)
86	)
88	SVC events

8A	Program Interrupt events
8C	Program Manipulation events
8E	Event trap

9 APPENDIX III: LEVEL1 PROGRAM INTERRUPTS

The following table lists the Program interrupt codes that can occur on LEVEL1:

CODE	DESCRIPTION
2	Operation
8	Arithmetic Overflow
A	Conversion Overflow
C	Division by Zero
E	Alignment
10	Register Specification
12	Privilege
14	Stack Overflow
16	Stack Underflow
18	Execute
20	No Free Memory
22	Invalid FREE instruction
24	Invalid CPC
26	Zero S/360 UCB Address

10 APPENDIX IV: EXTENDED INSTRUCTION OP CODES

The following table lists the operation codes for the LEVEL0 extended instructions:

INSTRUCTION	CODE
ENT	BE
EXCP	FD
FREE	43
FREER	03
GET	41
GETMAX	02
GETR	01
INTB	6F
POST	FE
QTIMER	64
RET	0B
SIGNAL	FC
WAIT	7E
WRITE	67

11 APPENDIX V: M4ALIB MACROS

Certain macros are provided for the LEVEL1 assembly language programmer. These macros reside in the BUGS macro library (M4ALIB) and are described in the following paragraphs.

11.1 EVENT

This macro generates an Event entry which can be placed on an EVL. It is coded as follows:

```
[label]   EVENT    link,name,flags,entry-point
                   [,priority,stack-frame-size]
```

The priority and stack frame size need only be coded for parallel routines.

11.2 AUTO/ENDAUTO

These two macros are used in each routine to generate a DSECT describing the stack frame section. The DSECT is begun by coding:

```
label     AUTO
```

which generates:

```
          LINE
*
*                    AUTOMATIC STORAGE MAP
*
          LINE
label     DSECT
          USING     label,SFP
labelP    DS        A                    PREVIOUS POINTER
labelN    DS        A                    NEXT POINTER
labelR    DS        15H                  REGISTER SAVE
labelA    DS        0C                   AUTOMATIC STORAGE
```

Following the AUTO, DS's for the automatic variables can be coded. Once all the automatic space is defined, the programmer should code:

```
         ENDAUTO
```

which will generate:

```
labelL   EQU      *-labelA              AUTO STORAGE LENGTH
&SYSECT  CSECT
         LINE
```

to end the DSECT. The symbol "labelL" should be used in the ENT instruction to specify the length of the automatic storage desired. Do not forget to include space for status data.

11.3 RETCODE

This macro is used in a subroutine to return a code in one of the registers of the invoking routine. It is coded:

```
[label] RETCODE invoker-reg[,code | (reg) ]
```

The return code specified in the macro or contained in the (reg) register is placed in the register of the invoking routine specified by "invoker-reg". Register 2 is bashed in the process.

11.4 CPC

This macro is used to generate a CPC for use with the EXCP instruction.

```
[label] CPC command,flags[,address[,length]]
```

If the address and/or length is not coded, these fields are set to zero in the generated CPC.

11.5 LEVEL0

This macro generates an equate table of the LEVEL0-defined locations in lower memory accessable by LEVEL1. A listing of the macro is available for those who need it.