| B U G S |

Brown University Graphics System

LEVEL0 Extended Machine[1]

A Program Logic Manual

Russell W. Burns

The Brown University Graphics Project

Division of Applied Mathematics

Box F

Brown University

Providence, Rhode Island 02912

September 15, 1976

---

[1]This research is being supported by the National Science Foundation Grant GJ-28401X, the Office of Naval Research, Contract N00014-67-A-0191-0023, and the Brown University Division of Applied Mathematics; Principal Investigator Andries van Dam.

Abstract

LEVEL0 is the nucleus program of the BUGS extended machine. This document describes the logic and organization of LEVEL0. It is intended for use by systems programmers debugging and modifying the operating system as a guide to the code. A thorough knowledge of the META 4A Principles of Operation and the LEVEL0 Extended Machine Principles of Operation is assumed.

TABLE OF CONTENTS

1 Introduction...1

2 The Supervisor...2
2.1 Dispatcher...2
2.2 Program Checks...2
2.3 Extended Instructions......................................3
2.3.1 Execution Control Instructions...........................3
2.3.1.1 SIGNAL: The Scheduler..................................3
2.3.1.2 WAIT: Suspend Execution................................4
2.3.1.3 POST: Continue Execution...............................5
2.3.1.4 ENT: Subroutine Entry..................................5
2.3.1.5 RET: Subroutine Exit...................................6
2.3.2 Data Management..7
2.3.2.1 GET: Get Controlled Storage............................7
2.3.2.2 GETMAX: Get Maximum Controlled Storage.................8
2.3.2.3 FREE: Free Controlled Storage..........................9
2.3.3 Interprocessor Communications...........................10
2.3.3.1 INTB: Interrupt META 4B...............................10
2.4 Control Blocks..10
2.4.1 Stack Frame Header......................................10
2.4.2 Stack Frame...12
2.4.3 CPU Unit Control Blocks.................................13
2.4.3.1 A-CPU...13

3 I/O Supervisor..14
3.1 Register Usage..14
3.2 Dispatcher..14
3.2.1 EXCP Routines...15
3.2.1.1 COMSIO..15
3.2.1.2 COMRDY..15
3.2.2 Interrupt Routines......................................16
3.2.3 META 4B Interrupt Processing............................16
3.3 EXCP Processing...17
3.4 CPC Processing..17
3.4.1 Unit Dependent Routines.................................18
3.5 I/O Interrupt Processing..................................18
3.6 Timer Control...19
3.6.1 Timer Interrupt Routine.................................19
3.6.2 QTIMER Instruction......................................19
3.6.3 UPTIMER Routine...20
3.7 Control Blocks..21
3.7.1 Unit Control Block......................................21
3.7.1.1 BUGSUCB Macro...22
3.7.1.2 Timer UCB Extension...................................23

1 INTRODUCTION

The standard LEVEL0 module is assembled from two SYSIN files, LEV0SUP [CSECT LEVEL0, entry points LEVEL0, DISPATCH, SIGNALEV] and LEV0IOS [CSECT EXCP, entry points EXCP, ZLIOH, UPTIMER, QTIMER, and the Unit Control Blocks]. Functions of the extended machine are roughly diveided into supervisor functions, in LEV0SUP, and I/O supervisor functions, in LEV0IOS. Both source files reside on the Graphics archival disk, and the text in BUGSLIB TXTLIB on RWB.

The version of LEVEL0 for the Null B is no longer being upgraded. It consists of one CSECT in two SYSIN files, LEV0-1 and LEV0-2 on the Graphics archival disk. Although the supervisor functions are largely the same, the I/O supervisor has been extensively re-written in the process of deleting the Null B support.

The differences in the LEVEL0 for Dynamic Relocation of Programs and Data will be discussed in another document[2]

---

[2] ?

2 THE SUPERVISOR

The LEVELO supervisor handles user requests for execution control and core management. It is entered upon program checks and supervisor call interrupts. The supervisor operates in supervisor mode, with I/O interrupts disabled. Global register usage is as follows:

R12 --> user's stack frame header
R15 --> user's current save area

Supervisor calls are merely reflected up to LEVEL1 by signalling the appropriate event name.

2.1 DISPATCHER

Entry: DISPATCH

LEVELO Dispatch is a simple priority dispatcher, entered only when a return to a user would be inadvisable, or when a new task is contending for the CPU. It searches the stack frame queue for the highest priority stack frame which is not in a wait state, and dispatches it. If all stack frames are in wait state (MSR bit 6 on), Dispatch calls UPTIMER[3] to compute the running CPU time and then dispatches the top stack frame, placing the machine in a wait state.

2.2 PROGRAM CHECKS

Entry: PGMCHK

The user communicates his requests for LEVELO services through extended instructions, which cause operation exceptions. The LEVELO program check handler reflects all other program checks up to LEVEL1 by signalling the appropriate event name (X'D0xx').

---

[3] vide Timer Control, Section 3.6

- If the reason for entry was an operation exception, the program check handler searches for the offending opcode in the extended opcode table. Upon success, the program check handler calls the associated routine via R13, and then returns to the user, unless the routine exits to the dispatcher.
- Otherwise, an invalid opcode event is signalled up to LEVEL1.

2.3 EXTENDED INSTRUCTIONS

The Extended Instruction Handlers are entered either through the program check handler or through internal call. Both the internal calling sequence and the extended instruction format will both be described where appropriate.

2.3.1 EXECUTION CONTROL INSTRUCTIONS

The Execution Control Instructions provide the scheduling, blocking, creation and destruction of tasks in BUGS. As such, they are intimately related to the dispatcher and to each other.

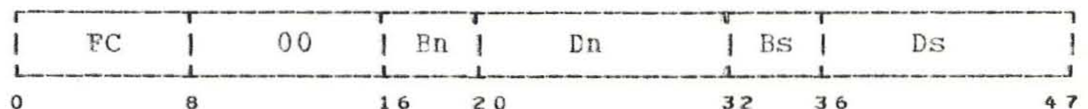
2.3.1.1 SIGNAL: The Scheduler

Entry: SIGNALEV

Internal Calling Sequence:

R8 contains event name
R9 --> status information
R10 contains length of status information

Instruction Format:



FC	00	Bn	Dn	Bs	Ds
0	8	16	20	32 36	47

The SIGNAL routine acts as the scheduler of interrupt processing. It searches the appropriate event list for a matching entry. If none is found, the low order hex digits of the event name are successively zeroed and the list re-searched. If no entry matches even partially, the trap event entry is chosen.

In any case, there are three possibilities of event list entry types which could be selected:

- If the entry indicates the event should be ignored, SIGNAL exits to the Dispatcher.
- If the entry indicates the event sould be handled syncronously (immediate event), SIGNAL exits to ENT processing to start the routine running immediately.
- Otherwise SIGNAL creates a new stack frame header, copying the entry point,priority, and stack frame size from the event entry and exits to the Dispatcher.

2.3.1.2 WAIT: Suspend Execution

Entry: WAIT

Instruction Format:

7E	00	Xw	Bw	Dw
0	8	12	16	20 31

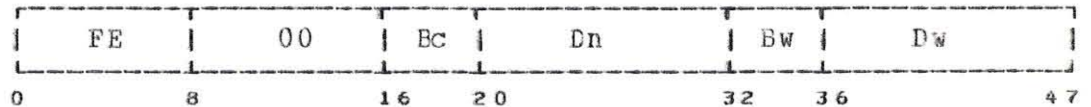
The WAIT extended instruction checks the high order bit of the Wait Control Halfword:

- If the bit is on, WAIT returns to the caller. (via R13).
- If the bit is off, WAIT stores the caller's stack frame address in the WCH, sets the wait bit on in the stack frame MSR, and exits to the dispatcher.

2.3.1.3 POST: Continue Execution

Entry: POST

Instruction Format:



The POST extended instruction swaps the former contents of the WCH with the return code; it then checks the former contents of the WCH:

- If the former contents were zero, POST returns to the caller (via R13).
- If the contents were non-zero, it is assumed to be a stack frame pointer. It wait bit in the MSR of that stack frame is turned off, and POST returns to the caller (via R13).

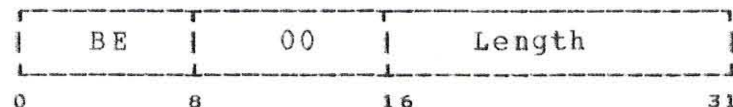
2.3.1.4 ENT: Subroutine Entry

Entry: SYNCHENT

Internal Calling Sequence:

R2 contains length of automatic storage

Instruction Format:



The ENT routine computes the total amount of stack storage required by the user for linkage, save area and his automatic data:

- If the requirement can be met from the current extension, the new remaining length is calculated and stored.

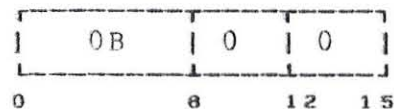
- Else a stack frame pointer is set in R15 for the user, and chained into the stack.

- If bit 3 is on in the MSR, SYNCHENT was entered due to an interrupt; the status information is moved into the stack and ENT exits directly to the new routine.

- Else the PC in the interrupted stack frame is set to the former contents of R14, R2 through R14 are loaded from the interrupted stack frame, and ENT exits directly to the new routine.

2.3.1.5 RET: Subroutine Exit

Instruction Format:



The RET extended instruction handler sets R15 equal to the address of the previcus savearea:

- If there is no previous savearea, the stack frame header is removed from the active queue and freed, and RET exits to the dispatcher.

Else it computes the new remaining length:

- If the address of the next savearea in the savearea pointed to by R15 is the same as the former contents of R15, RET stores the new remaining length.

- Else it is popping up from a stack extension, and frees the extension.

- If the new remaining length is equal to the total length, the header is dequeued, the stack frame freed, and RET exits to the dispatcher.

RET then checks the status of the routine to which control is about to be returned:

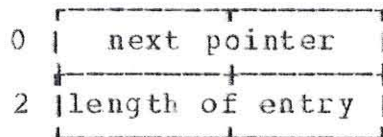
- If it is in a wait state, RET exits to the dispatcher.

- Else it returns directly to the new stack frame.

2.3.2 DATA MANAGEMENT

The free memory is maintained in a linked list, with the pointer to the starter element in the META 4A Unit Control Block. If the low order bit of this semaphore is one, the META 4B is using the queue, and the META 4A must loop, waiting for the list to be free.

The entries are stored by ascending address, to facilitate concatenation of adjacent free areas. Each entry is of the format:



next pointer: pointer to the next element on the free queue.

length of entry: the length of this entry in bytes, including the header.

2.3.2.1 GET: Get Controlled Storage

Entry: GETMEM

Internal Calling Sequence:

R6 contains amount of storage required (in bytes)

Returns:

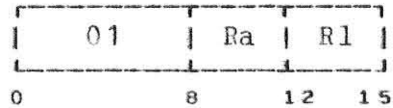
R5 --> new allocated area
R6 contains rounded length

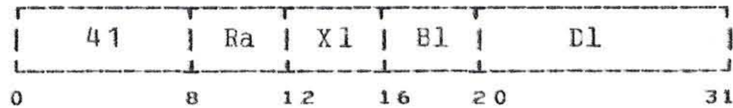
Exits:

- normal -- to caller via R14

* abnormal -- to PGMCHK

Instruction Format:





The GETMEM routine searches the free memory queue for the first entry long enough to satisfy the request:

* If no entry long enough is found, GETMEM exits to PGMCHK to signal a No Free Memory program check.

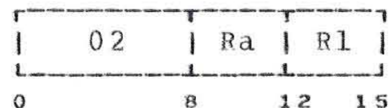
* If the entry matched the requested length exactly, it is dequeued and its address returned to the caller.

* If the entry is longer than the request, its length is decremented by the requested length, and the address of the entry plus the length remaining is returned.

2.3.2.2 GETMAX: Get Maximum Controlled Storage

Entry: GETMAX

Instruction Format:



The GETMAX routine searches the free memory queue for the largest element:

* If the queue is null, GETMAX exits to PGMCHK to signal a No Free Memory program check.

* Else it dequeues the entry found and returns its address to the user, exiting via R13.

2.3.2.3 FREE: Free Controlled Storage

Entry: FREEMEM

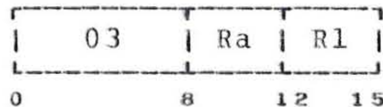
Internal Calling Sequence:

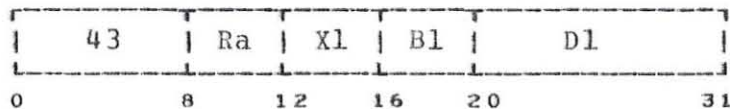
R6 --> area to free
R7 contains length of area to free

Exits:

* normal -- to caller via R14
* abnormal -- to PGMCHK

Instruction Format:





FREEMEM searches the free memory queue for an entry with a forward pointer greater than the address to be freed:

* If unsuccessful, it selects the last element on the list.
* It then checks to see if the end address of the selected element is adjacent to the area to be freed.
* If so, the length of that element is incremented by the length to be freed.
* Else, the new element is enqueued at that point.

FREEMEM then examines the next element on the queue:

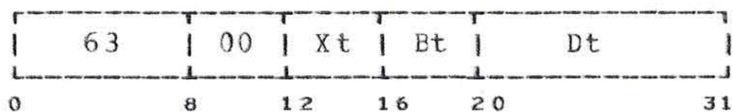
* If it is adjacent to the new element, it is dequeued and the new element's length adjusted accordingly.

2.3.3 INTERPROCESSOR COMMUNICATIONS

2.3.3.1 INTB: Interrupt META 4B

Entry: SHOULDER

Instruction Format:



63	00	Xt	Bt	Dt
0	8	12	16	20 31

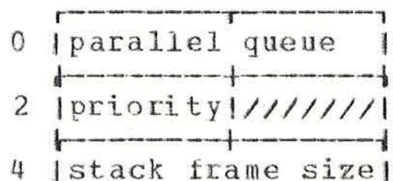
The INTB extended instruction handler issues a start I/O to the IPI to determine if the last interrupt has been acknowledged by the META 4B:

* If the last interrupt is still pending, the routine loops until the META 4B acknowledges the interrupt.

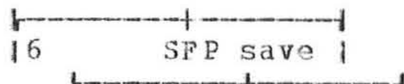
* Otherwise, it stores the interrupt code in the META 4A UCB Unit Status Halfword, and issues a start I/O to interrupt the META 4B.

2.4 CONTROL BLOCKS

2.4.1 STACK FRAME HEADER



Offset	Field	
0	parallel queue	
2	priority	////////
4	stack frame size	



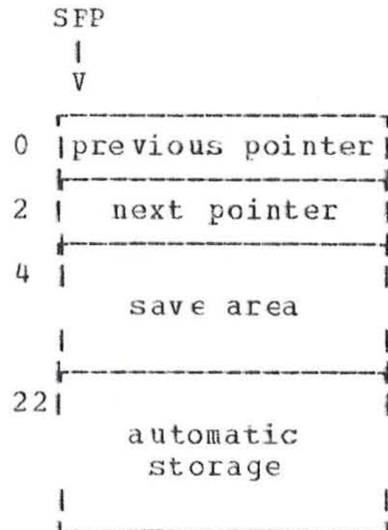
parallel queue: this pointer is used by LEVEL0 to maintain a queue of the stack frames of each of the parallel routines currently running on the system. The head of this queue is in memory location X'60'.

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.

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

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

2.4.2 STACK FRAME



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.

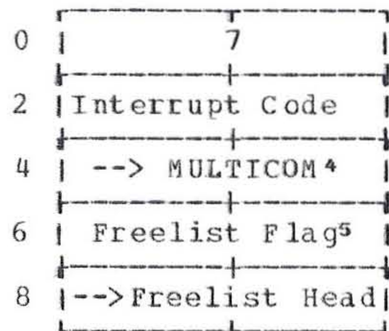
save area: these 15 halfwords are used to store the routine's MSR through register 14 at any time his execution is delayed due to an actual machine interrupt or to a subroutine call.

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.

2.4.3 CPU UNIT CONTROL BLOCKS

The CPU Unit Control Blocks are used for interprocessor communications and vary in format from the I/O unit UCB's.

2.4.3.1 A-CPU



---

[4]Initialized by MULTIPAC at execution time.
[5]Low order bit used as Freelist semaphore.

3 I/O SUPERVISOR

The I/O Supervisor handles all local I/O interrupts, and processes the EXCP extended instruction. Since the supervisor simulates a virtual channel, the EXCP and interrupt processing are intimately related.

3.1 REGISTER USAGE

Some of the general purpose registers have fixed contents during operation of the I/O Supervisor:

R3 --> current CPC
R10 --> Unit Control Block
R15 --> user's current save area

3.2 DISPATCHER

Entry: EXCPDISP

Internal Calling Sequence:

R10 --> UCB
R5 contains current operation

The possible operations are:

- 0 -- EXCP
- 1 -- READ
- 2 -- WRITE
- 3 -- NOP
- 4 -- SENSE
- 5 -- SENSE with reset
- 6 -- SPECIAL

• 7 -- Interrupt

The dispatcher uses the code in R5 as an index into the Where to Go table in the UCB. It then uses the byte at that offset in the UCB WTG table to index into the global WTG table. Since many of the entries, e.g. NOP, SENSE, SENSE with reset, are identical, this saves space from individual WTG tables.

3.2.1 EXCP ROUTINES

The EXCP routines are generally responsible for insuring that the unit is ready before starting an I/O operation.

Register Usage:

```
R3   --> current CPC
R10  --> UCB
R13  --> return point
R15  --> user's current save area
```

3.2.1.1 COMSIO

Entry: COMSIO

The common start I/O routine issues a sense to the unit:

- If the USH returned indicates that the unit is busy or offline, COMSIO sets the condition code appropriately in the user's stack frame and exits via R13.
- Else COMSIO continues into COMRDY.

3.2.1.2 COMRDY

Entry: COMRDY

COMRDY is entered from COMSIO or directly from the EXCP Dispatcher, if the device cannot be offline. It

sets the "logically busy flag on in the UCB and calls CPC processing[6] via R9.

3.2.2 INTERRUPT ROUTINES

Register Usage:

R3 --> current CPC
R10 --> UCB
R15 --> interrupted task's current save area

The interrupt routines are completely unit dependent. They do share some common characteristics:

- They check for errors, and either retry the operation or set the device non-busy and exit to SIGNALEV to signal an error event.
- Else they check for operation complete:
- If the operation on the CPC is complete, they call CPC processing via R9 at entry COMINTEN to continue processing on the CPC chain. When control is restored, they exit to SIGNALEV to signal the operation complete event.
- Else, they issue the next start I/O in the sequence and exit to the interrupted task.

3.2.3 META 4B INTERRUPT PROCESSING

When the I/O Supervisor recognizes an interrupt from the META 4B, it retrieves the interrupt code from the USH in the META 4B UCB, acknowledges the interrupt, and exits to SIGNALEV to signal event X'1031', with the interrupt code as its status.

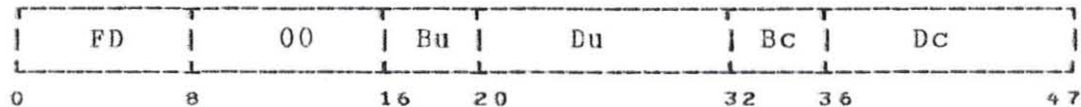
---

[6]vide Section 3.4

3.3 EXCP PROCESSING

Entry: EXCP

Instruction Format:



The EXCP routine sets R10 to pointer to the proper UCB, sets R3 to point at the user's CPC, then tests the UCB flags:

- If the unit is logically busy, it sets the condition code appropriately in the user's stack frame and exits via R13.

- Else it enters the EXCP dispatcher with a 0 in R5.

3.4 CPC PROCESSING

Entry: COMINTRD

Register Usage:

```
R3   --> current CPC
R9   --> return point
R10  --> UCB
R15  --> user's current save area
```

COMINTRD is the common CPC interpreting loop. It stores the current CPC address in the UCB, picks the opcode from the CPC, clears out the high order four bits, and places a copy of them in R14:

- If the opcode is greater than 7, it exits to SIGNALEV to signal an Invalid CPC program check (X'D024').

- If the opcode is equal to 7 (TIC), it picks up the new CPC address and goes back to the start of the loop.

- Else it calls EXCPDISP via R4, with the CPC opcode in R5 to call the unit dependent routine.

When it returns, it checks the immediate operation flag in the UCB:

* If it is off, an interrupt is expected, and COMINTRD returns to the caller via R9.

* If it is on, COMINTRD checks for chaining in the CPC flags:

* If chaining is indicated, COMINTRD bumps R3 by 6 and re-enters the loop.

* Else it resets the "logically busy" flag in the UCB and exits via R9.

3.4.1 UNIT DEPENDENT ROUTINES

Register Usage:

```
R3    --> current CPC
R4    --> return point
R9    RESERVED
R10   --> UCB
R14   contains modifier field of CPC
R15   --> user's current save area
```

The unit dependent routines generally start an I/O operation, set a flag in the UCB to indicate what operation is in progress and then exit via R4:

* If the operation completes immediately, the immediate operation flag is set in the UCB flags.

3.5 I/O INTERRUPT PROCESSING

Entry: ZLIOH

ZLIOH sets R10 and R3 to point to the interrupting unit's UCB and current CPC respectively. It then checks the I/O old MSR:

* If the interrupted task was in a wait state, it sets the expected CPU interval equal to the current contents of the timer.

It then checks the interrupting unit's address:

- If the interrupt was from the META 4B, it goes to the META 4B interrupt handler[7]

- Else it branches to EXCPDISP with a 7 in R5, indicating an interrupt.

3.6 TIMER CONTROL

The Timer control code allows the LEVEL1 user to keep track of running and CPU time, and to set time intervals.

3.6.1 TIMER INTERRUPT ROUTINE

Entry: GRANDDAD

The Timer Interrupt Routine calls UPTIMER to update the CPU and Running times. It then sets a new time interval in the Timer, and checks to see if a time interval set by the user has expired:

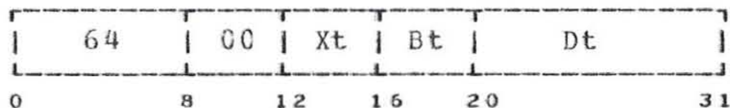
- If the user's interval has not expired, it returns directly to the interrupted routine.

- Else it exits to SIGNALEV to signal event X'2001'.

3.6.2 QTIMER INSTRUCTION

Entry: QTIMER

Instruction Format:



The QTIMER routine calls UPTIMER to get the up to date time, then moves the RUNNING and CPU times from the UCB

---

[7] vide Section 3.2.3for details.

extension to the user's area, and exits directly to the user.

3.6.3 UPTIMER ROUTINE

Entry: UPTIMER

Exits: to caller via R13

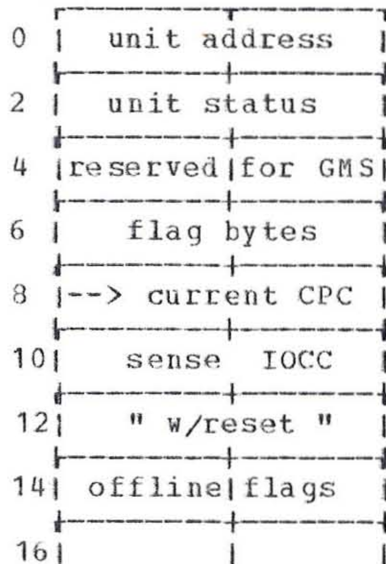
UPTIMER is called by the dispatcher wchen it is about to go into a wait state; it is called by the Timer Interrupt Routine to update the clocks; and it is called by QTIMER to get the corrected times.

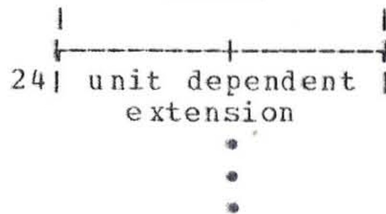
It subtracts the contents of the Timer from the expected interval, then adds the computed value to the 32 bit clocks. It then replaces the expected interval with the contents of the Timer. This code, combined with the setting of the CPU expected interval by the I/O interrupt handler should maintain the CPU and Running time 32 bit clocks.

3.7 CONTROL BLOCKS

3.7.1 UNIT CONTROL BLOCK

The Unit Control Block contains the device independent information for each device and most of the residual control information needed to simulate the virtual channel:





unit address: virtual unit address.

unit status: USH from last SIO or interrupt.

reserved for GMS: unused in all UCB's except for disk, where it is used for Dr. Memory communications.

flag bytes: residual control bits:
byte 0:
bit 0: logically busy; set at start of EXCP; reset at operation complete

bit 1: immediate; set if CPC operation was immediate.
bit 2: verify (disk only); cylinder address must be verified before write.
bit 3: multisector (disk only); multisector operation in progress.
bits 4-7: unused.

byte 1:
bit 0: write; write operation in progress.
bit 1: read; read operation in progress.
bit 2: verifying (disk only); cylinder address is being verified.
bit 3: control; control operation in progress.
bits 4-7: unused.

3.7.1.1 BUGSUCB Macro

The BUGSUCB macro is used to generate the Unit Control Blocks and to generate a DSECT of the UCB's:

```
label      BUGSUCB  addr,offline,sense,senser,
                    (wtg-list)[,UNITX=size]
```

label: name of UCB.

addr: virtual unit address

offline: self defining term for bits which if set indicate that the device is busy or offline.

sense: hex code for 2nd half of sense ICCC.

senser: hex code for 2nd half of sense w/reset IOCC.

wtg-list: names of pointers in global Where to Go Table for device dependent routines.

size: number of halfwords for unit dependent extensions to UCB.

If TYPE=DSECT is specified, a DSECT rather than a real UCB will be assembled.

3.7.1.2 Timer UCB Extension

The UCB device dependent extenstion for the Interval Timer cortains the running clocks and the information neccessary to control them.

Offset	Field
24	time precision
26	time interval
28	RESERVED
30	CPU interval
32	running
	time
36	CPU
	time
40	expiration
	time

timer precision: time interval to be set when Timer expires.

time interval: amount of time expected to have passed when a Timer interrupt occurs.

CPU interval: amount of CPU time expected to have passed when a Timer interrupt occurs.

running time: time in timer units since IPL.

CPU time: CPU time in timer units since IPL.

expiration time: time at which LEVEL1 wishes to receive an interrupt.