PRIME

PRIMOS Operating
System Specialist

Revision 19.4

PRIMOS Operating System Specialist

Revision 19.4

Date: August, 1985

Revision: 5

All correspondence on suggested changes to this document should be directed to:

Prime Technical education Center
Prime Computer, Inc.
Prime Park
Natick, MA 01760

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT'D)

TABLE OF CONTENTS (CONT'D)

TABLE OF CONTENTS (CONT'D)

Section 1 - Hardware Features

Objectives: The student will be able to

- o describe the peripherals and controllers on a Prime system.
- o describe the major components of the CPU.
- o describe the contents and use of the register file groups

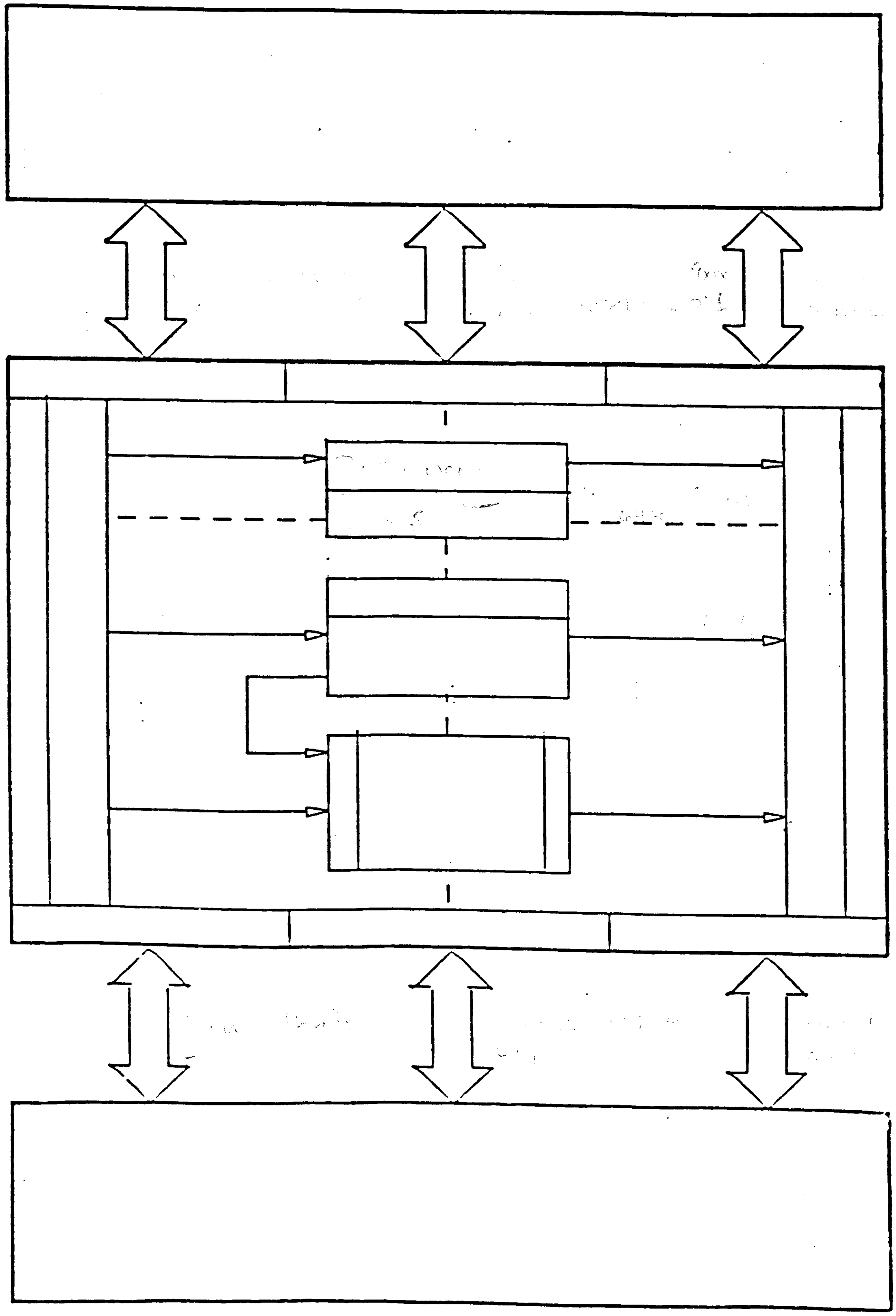
PRIMOS OPERATING SYSTEM

The chief features of the Primos operating system are:

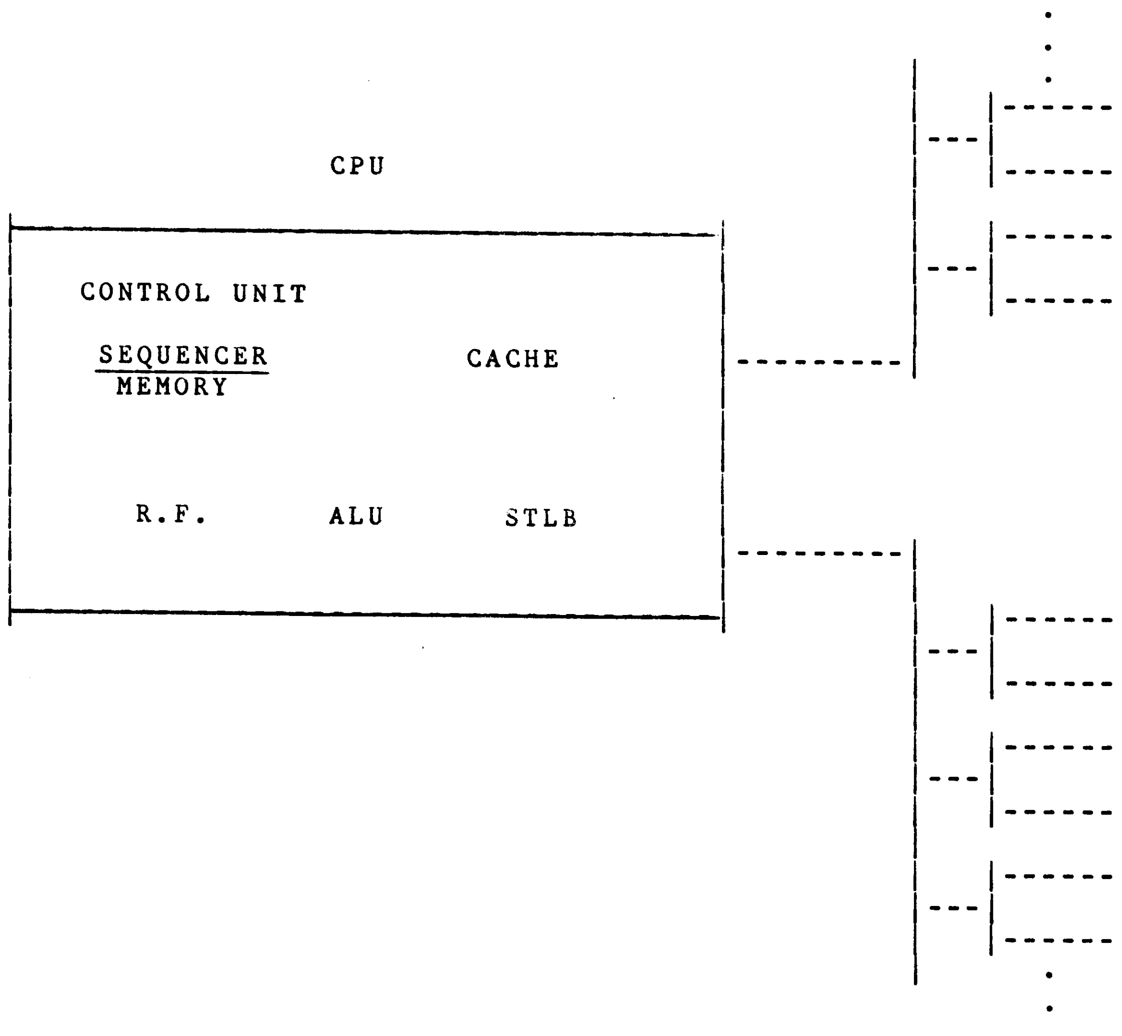
1. INTERACTIVE - up to 255 user processes
   (14+ interrupt processes)
2. 64 MB maximum private virtual address space per user
3. Users share the resources of the system

   High speed memory
   Programmable Interval Clock
   Peripherals and controllers
      System Console
      Disk Drive(s)
      AMLC(s)/ICS1(s)/ICS2(s)
      SMLC(s)/MDLC(s)
      Ring Node Controller (PNC)
      Magnetic Tape Drive(s)
      Line Printer(s)

Microcode-Based CPU



CENTRAL PROCESSOR UNIT



REGISTER FILE

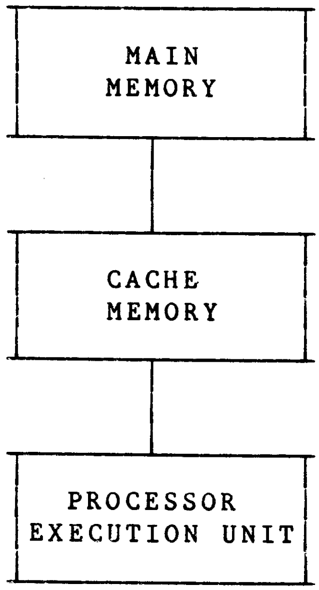
	MICROCODE SCRATCH HIGH	MICROCODE SCRATCH LOW	DMA HIGH	DMA LOW	CURRENT REGISTER HIGH	CURRENT REGISTER LOW
0					GR0:OLT2	
1					GR1:PTS	
2					GR2(1,A,LH)	(2,B,LL)
3					GR3 (EH)	(EL)
4					GR4	
5					GR5 (3,S,Y)	
6					GR6	
7					GR7 (0,X)	
10					FAR0 (13)	
11					FLR0	
12					FAR1/FAC(4)	(5)
13					FLR1/FAC(6)	
14					PB	
15					SB (14)	(15)
16					LB (16)	(17)
17					XB	
20					DTAR3 (10)	
21					DTAR2	
22					DTAR1	
23					DTAR0	
24					KEYS	MODALS
25					OWNER	
26	LREGSET	CHKREG			FCODE (11)	
27	DSWPARITY				FADDR	(12)
30	PSWPB				CPU TIMER	
31	PSWKEYS				MICROCODE SCRATCH	
32	PPA:PLA	PCBA			"	
33	PPB:PLB	PCBB			CPNUM	
34	DSWRMA				"	
35	DSWSTAT				"	
36	DSWPB				"	
37	RSAVPTR				"	

Section 2 - Memory Management

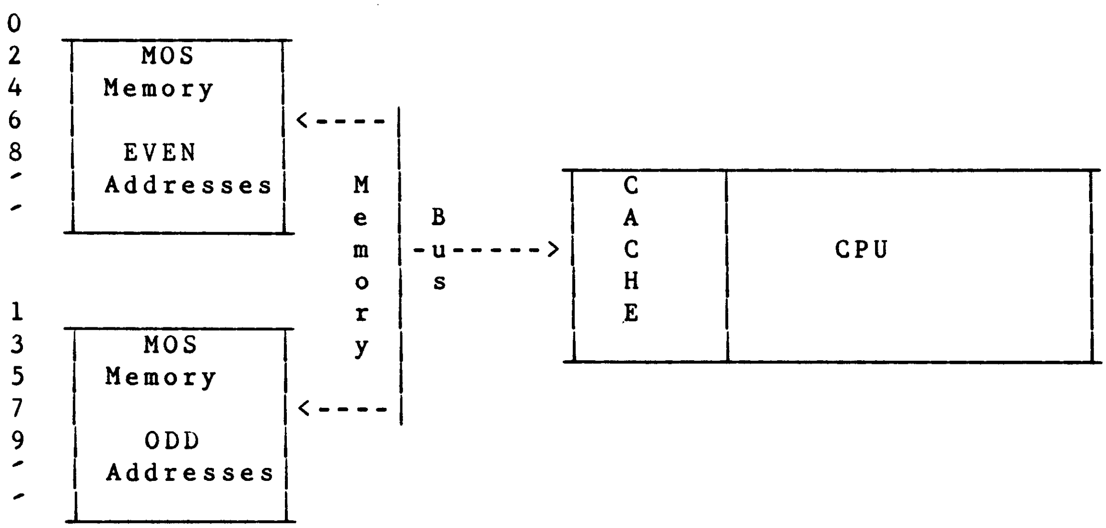
Objectives: The student will be able to

- describe how cache reduces the effective memory access time for memory reference instructions.
- explain how memory interleaving speeds up sequential memory access and increases the cache hit rate.
- distinguish between virtual and physical memory.
- describe the address translation hardware mechanism.
- describe how cache and the STLB are used to access a word of data.
- explain how a page fault is generated and handled.
- examine memory management-related variables and data structures in memory using VPSD.
- answer memory management-related questions by examination of source code.

CACHE FUNCTIONAL DIAGRAM



INTERLEAVING



SEGMENTATION

Virtual Memory is divided into variable length SEGMENTS (64K words max) 4096 SEGMENTS define 512 MB of Virtual Memory. The Virtual address space is divided into 4 areas (DTARs), each area consisting of 1024 ('2000) segments.

Segments	Area	CURRENTLY ENABLED
'7777 – '6000	PRIVATE PER USER (SYSTEM)	______
'5777 – '4000	PRIVATE PER USER (USER)	______
'3777 – '2000	SHARED BY ALL USERS	______
'1777 – '0000	EMBEDDED OPERATING SYSTEM	______

EFFECTIVE ADDRESS FORMAT

PROGRAM INSTRUCTIONS GENERATE AN EFFECTIVE ADDRESS (EA).

- 2 Bits RING NUMBER (defines privileges)
- 12 Bits SEGMENT NUMBER
- 16 Bits WORD NUMBER (within SEGMENT)

1	2 3	4	5 16	17 32
	RING		SEGMENT NO.	WORD NUMBER

The EFFECTIVE ADDRESS (28 BITS) is mapped to PHYSICAL MEMORY.

- 23 Bits of PHYSICAL ADDRESS
- Up to 16M Bytes of PHYSICAL MEMORY.
- 22 Bits PHYSICAL ADDRESS
- Up to 8M Bytes of PHYSICAL MEMORY.

RING NUMBER

There are 3 RINGS which define the privileges of access to the SEGMENT.

RING 0	is the most privileged and allows unrestricted access to all segments. Ring 0 is the only ring that can execute restricted instructions. PRIMOS runs in RING 0.
RING 1	Not currently used by software
RING 3	The least privileged. USERS run in RING 3.

Hardware defines access rights of:

Inner ring accessing memory in an outer ring.

Outer ring accessing memory in an inner ring.
GATE access

The SHARE command for DTAR 1

MEMORY MANAGEMENT TECHNIQUES

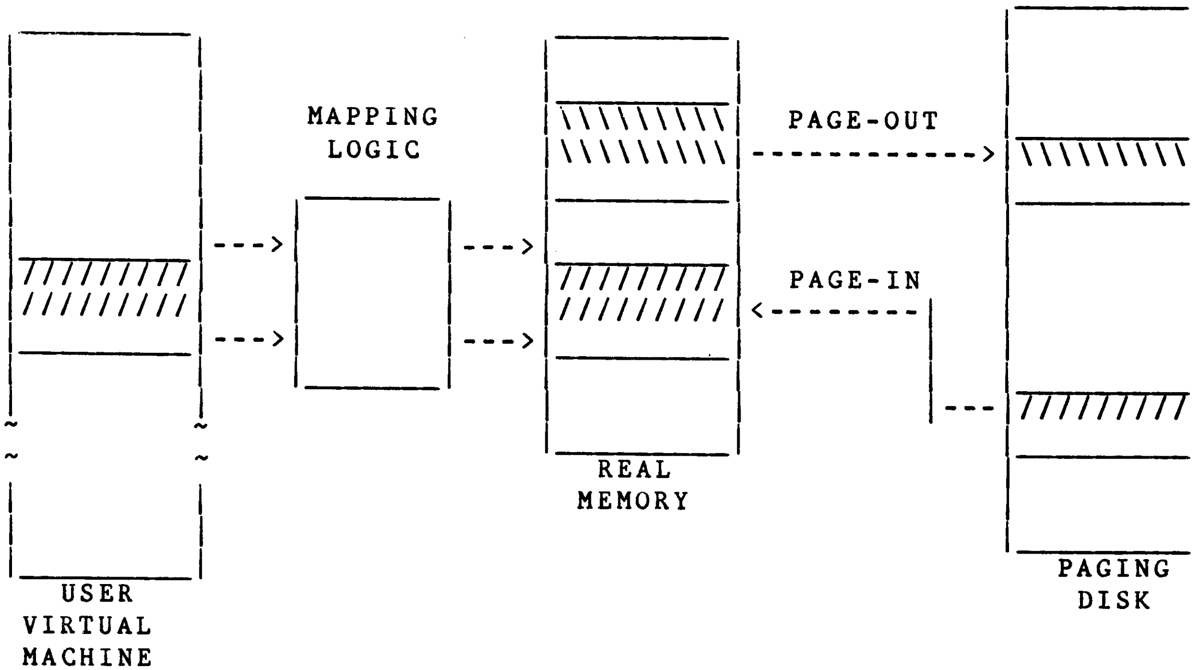
The total number of segments available is currently 8192.
All 8192 segments cannot be contained in physical memory.
Virtual Memory is divided into two parts:
1) the part in physical memory
2) the part on the paging disk

Certain information is too critical to be on the paging disk, it is "WIRED" ("LOCKED") into physical memory.
At COLD START, PRIMOS "wires" critical information, this area will grow as PRIMOS requires certain per-user data to be wired.
When user segments are allocated, paging space is allocated.

Programs generate VIRTUAL ADDRESSES.
The VIRTUAL ADDRESS is translated (mapped) to a main memory address.
If the required physical address is resident within physical memory, the access may proceed without interruption.
If not in physical memory, a PAGE FAULT will occur.

When a PAGE FAULT does occur, the program is suspended while the required page is moved from the PAGING DISK into main memory.
This is called PAGING IN.
If there is no physical memory page available, PRIMOS will use a Approximately-Least-Recently-Used algorithm to determine which page in physical memory will be PAGED OUT to allow space for the in-coming page.

MEMORY MANAGEMENT



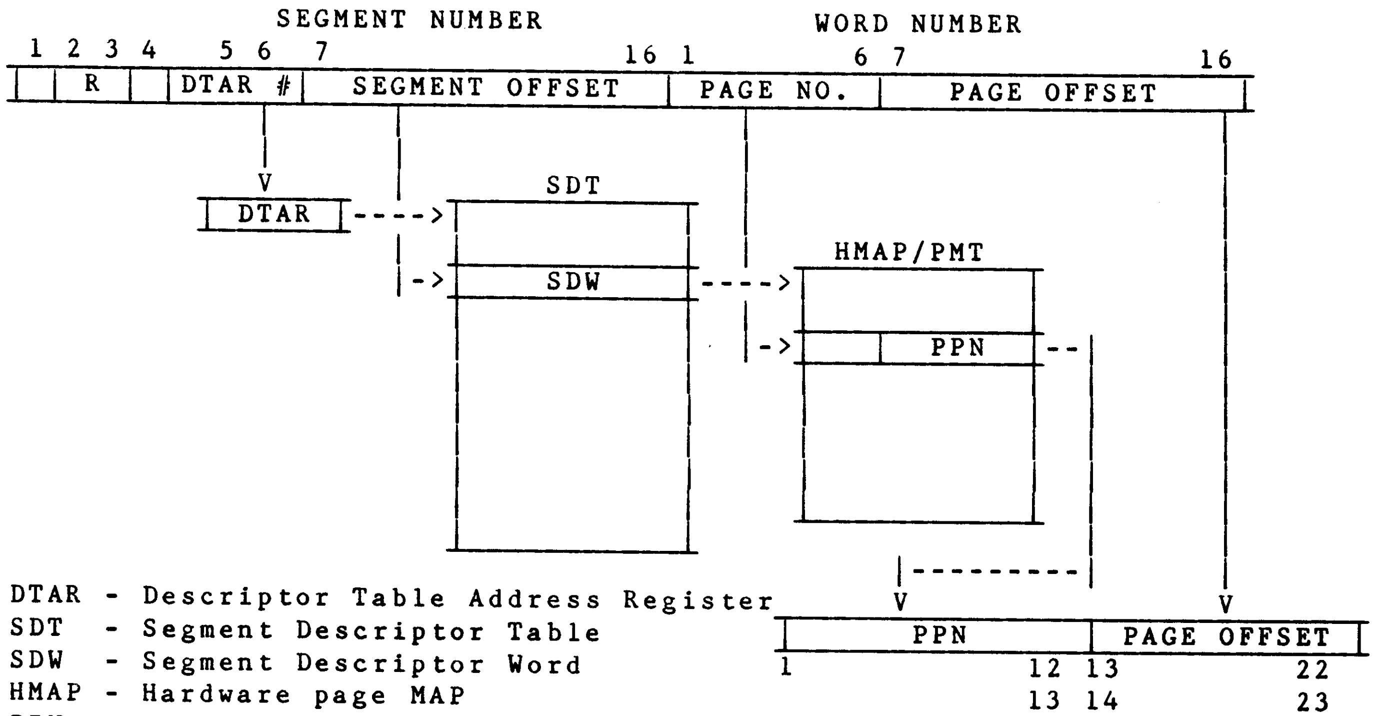
PAGE FAULT (Access then proceeds)

ADDRESS TRANSLATION

Every VIRTUAL ADDRESS is translated (mapped) to a physical address by accessing the STLB (Segmentation Translation Lookaside Buffer). The STLB holds the most recent virtual to physical address translations. When the STLB does not have a valid entry for the virtual address to be translated, hardware calculates the address translation using Descriptor Table Address Registers, Segment Descriptor Tables and Hardware Page Maps. The STLB is accessed again, this time being sure to get a STLB hit. During translation, a page fault will occur if the desired page is not in physical memory.

Simultaneous to the STLB access, hardware starts a CACHE access. If the word from cache is from the correct physical page, then the access is complete. If the word sought is not a valid cache entry, then the information is brought into cache from physical memory.

FULL ADDRESS TRANSLATION



DTAR - Descriptor Table Address Register
SDT - Segment Descriptor Table
SDW - Segment Descriptor Word
HMAP - Hardware page MAP
PPN - Physical Page Number

DTAR - DESCRIPTOR TABLE ADDRESS REGISTER

1	10	11 16
17	18	32

Bits 1-10 = 1024 minus number of entries in SDT
11-16 = High order 21 bits of physical address
18-32 of SDT origin
17 = must be zero

SDW - SEGMENT DESCRIPTOR WORD

```
 |1|     |     |10                 16|
 |F|A A A|B B B|C C C|               |
 17 18 20 21 23 24 26 27            32
```

```
Bits  27-32  =  Physical address of Page Map Table (HMAP)
       1-16     (Bits 11-16 must be zero)
         17  =  Fault Bit
      18-20  =  (AAA) Access rights from RING 1
                 000  no access
                 001  Gate access only
                 010  Read access only
                 011  Read and write access
                 100  reserved
                 101  reserved
                 110  Read and execute access
                 111  Read, write, and execute access
      21-23  =  (BBB) reserved for future use
      24-26  =  (CCC) Access rights from RING 3
                      same as RING 1 access bits
```

PAGE MAP ENTRIES - PAGE IN MEMORY

Processor supports > 8MB of physical memory

	1	2	3	4	5	6 7	8	9 ... 16
PMT	R	U	M	S	-	WIRE	F	DISK ADDRESS (HIGH)
	PHYSICAL PAGE NUMBER							

Processor supports <= 8MB of physical memory

	1	2	3	4	5 ... 16
HMAP	R	U	M	S	PHYSICAL PAGE NUMBER
LMAP	WIRE		F	DISK ADDRESS (HIGH)	

Resident bit is set when page is in physical memory.

Used bit is set by the address translation hardware.

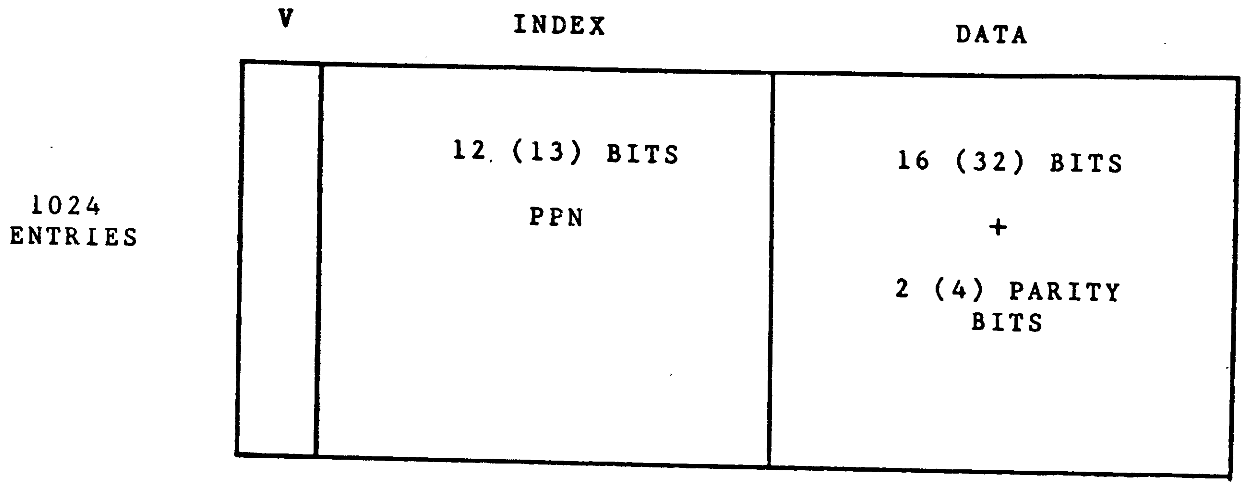
Modified bit indicates whether the page has been modified.

Shared bit is set to inhibit cache for all locations in this page.

WIRE bits are set to indicate this page is locked in physical memory.

PHYSICAL PAGE NUMBER is the physical address of the page.

THE CACHE


V
INDEX
DATA
1024
ENTRIES
12 (13) BITS
PPN
16 (32) BITS
+
2 (4) PARITY
BITS

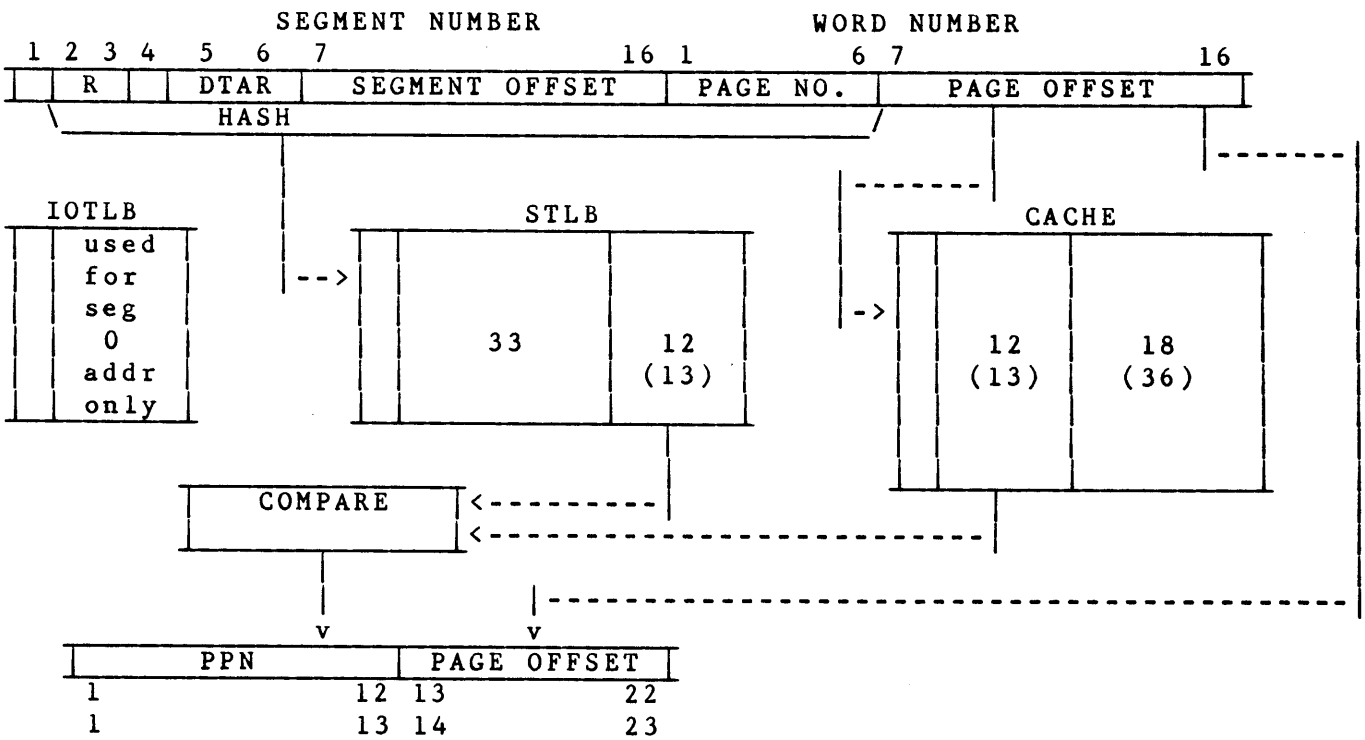
THE STLB

			Access Rights				
V	M	S	Ring 1	Ring 3	Process ID	Segment No.	Phys. Page No.
1 Bit	1 Bit	1 Bit	3 Bits	3 Bits	12 Bits	12 Bits	12 Bits (13)

THE IOTLB

V	PPN
1	12(13)

READ MEMORY ACCESS



PAGE FAULT

Whenever a user program issues a virtual address the hardware translates this address into physical memory using the STLB. An STLB ´miss´ may be caused by failure to find the desired entry, or by a reset valid bit for the desired entry. During full translation, the HMAP/PMT entry will indicate if the desired page is not in memory.

The page map entry contains a marker bit (bit 1) indicating whether or not the required page is held in memory. If the page is in physical memory, translation proceeds but if the page is not in memory, a PAGE FAULT occurs.

This fault causes a branch in execution through the user´s page fault vector to the fault table code. A CALF is then executed in the page fault catcher. (All page faults are handled by this routine).

The page fault catcher will:

1). Save the user state
2). Check recursive page fault. If so HALT
    Allow warm start but process takes fatal error.
3). Call PAGTUR

PAGE MAP ENTRIES - PAGE IN MEMORY

Processor supports > 8MB of physical memory

	1	2	3	4	5	6 7	8	9 16
PMT	R	U	M	S	-	WIRE	F	DISK ADDRESS (HIGH)
	PHYSICAL PAGE NUMBER							

Processor supports <= 8 MB of physical memory

	1	2	3	4	5 16
HMAP	R	U	M	S	PHYSICAL PAGE NUMBER
LMAP	WIRE		F	DISK ADDRESS (HIGH)	

Resident bit is set when page is in physical memory.

Used bit is set by the address translation hardware as well as by PAGTUR on a page-in, reset by PAGTUR aging the page.

Modified bit indicates whether the page has been modified.

WIRE bits are set to indicate this page is locked in physical memory.

First time in bit is set by PAGTUR on page-in, and reset by PAGTUR aging the page.

PAGE MAP ENTRIES - PAGE NOT IN MEMORY

Processor supports > 8MB of physical memory

	1	2	3	4	5	6	7	8	9	16
PMT	R	U	sta	S	tus	WIRE		F	DISK ADDRESS (HIGH)	
	DISK ADDRESS (LOW)									

Processor supports <= 8MB of physical memory

	1	2	3	4	5	6	16
HMAP	R	U	sta	S	tus	DISK ADDRESS (LOW)	
LMAP	WIRE		F	DISK ADDRESS (HIGH)			

Resident bit is reset when page is not in physical memory.

status is defined by bit 3 and bit 5 as follows:

00 not in, copy on disk
10 not in, no copy on disk
01 in transition, coming in
11 in transition, going out

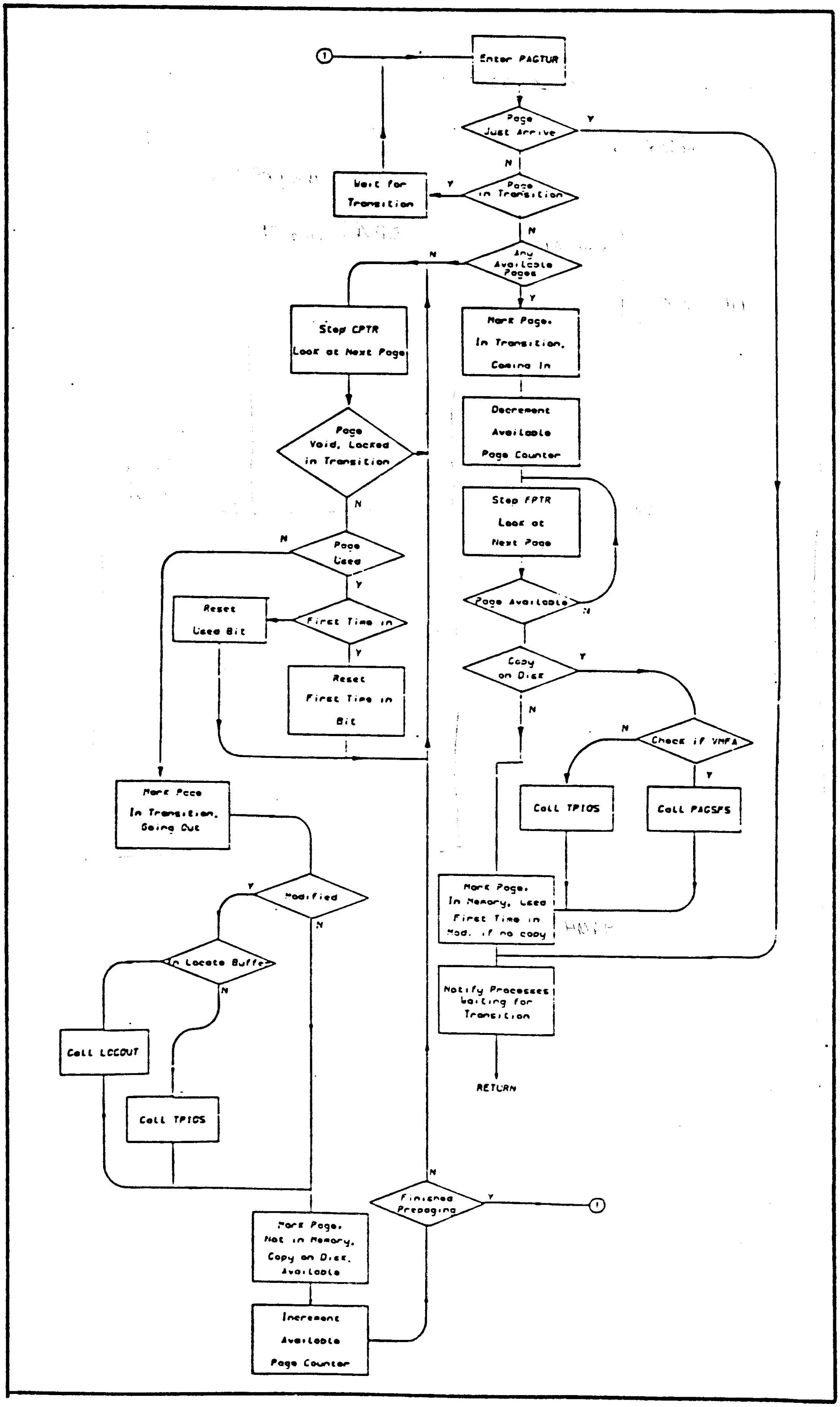
MMAP (segment 33)

1	2	3 ... 16
A	V	HMAP ENTRY SEGMENT NUMBER
		HMAP ENTRY WORD NUMBER
		DISK ADDRESS (LOW ORDER)

Available bit is set when this page is free for page-in.
Void bit is set to map out a missing or bad page.

```
                                          MMAP ENTRIES
          MMAP_STRT------>[                                        ]
five                      ~                                        ~
pointers  MMAP_CPTR------>[                                        ]
to                        ~                                        ~
MMAP      MMAP_FPTR------>[                                        ]
                          ~                                        ~
          MMAP_END8------>[                                        ]
                          ~                                        ~
          MMAP_END------->[                                        ]
```

```
MMAP_STRT   points to the first MMAP entry
MMAP_CPTR   is stepped during page-out
MMAP_FPTR   is stepped during page-in
MMAP_END    points to entry after last MMAP entry
MMAP_END8   If there are more than 8MB of memory
                points to last entry in the first 8MB
            else MMAP_END8 = MMAP_END
```

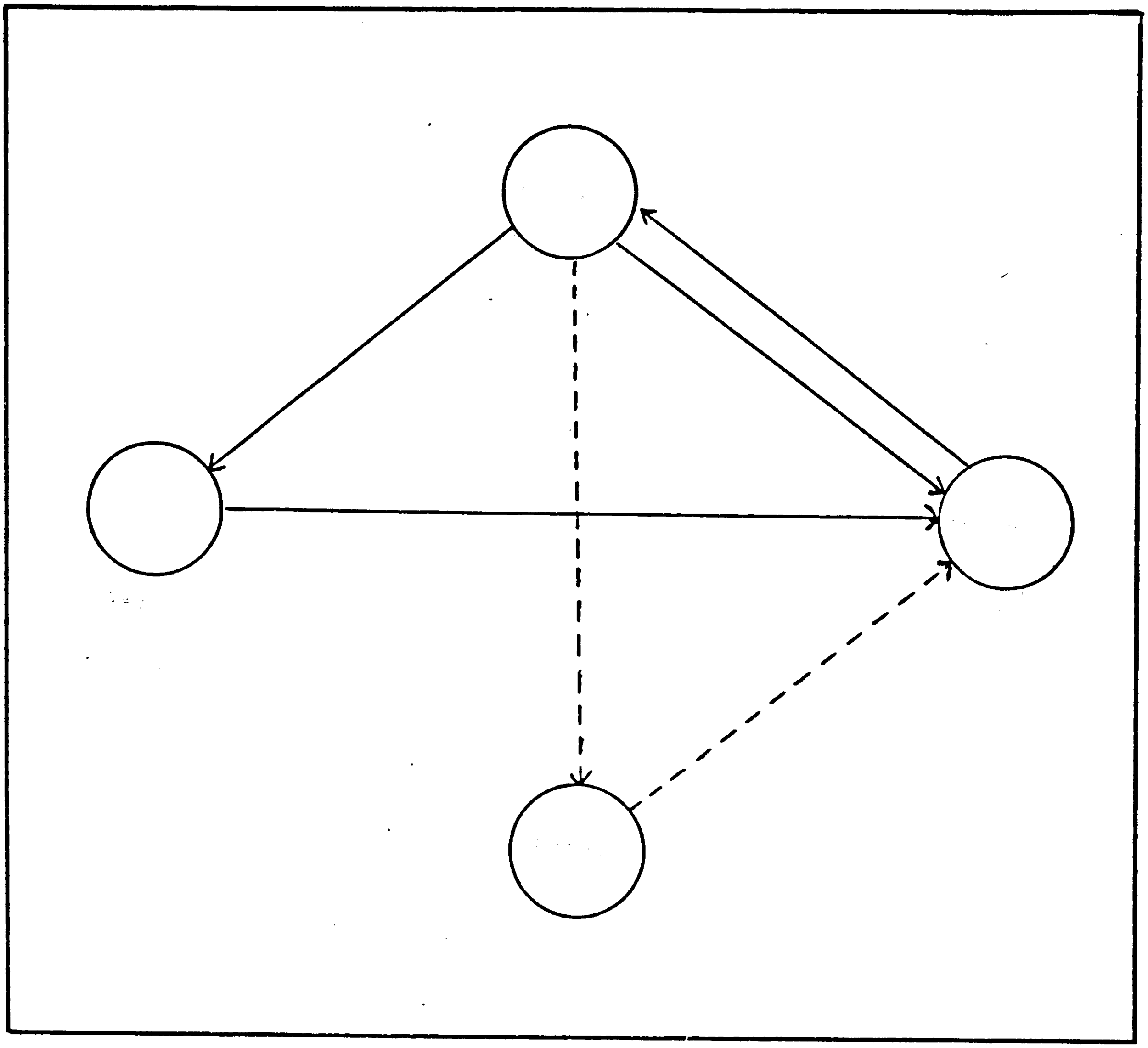

Enter PAGTUR
Page Just Arrive
Page in Transition
Wait for Transition
Any Available Pages
Step CPTR
Look at Next Page
Mark Page, In Transition, Coming In
Decrement Available Page Counter
Step FPTR
Look at Next Page
Page Available
Page Void, Locked in Transition
Page Used
First Time in
Reset Used Bit
Reset First Time in Bit
Copy on Disk
Check if VMFA
Call TPIOS
Call PAGSFS
Mark Page, In Transition, Going Out
Modified
In Locate Buffer
Call LCCOUT
Call TPIOS
Mark Page, In Memory, Used First Time in Mod. if no copy
Notify Processes Waiting for Transition
RETURN
Finished Prepaging
Mark Page, Not in Memory, Copy on Disk, Available
Increment Available Page Counter

Section 3 - Process Management

Objectives: The student will be able to:

- o describe the different process states.
- o describe the data structures and implementation of process exchange.
- o explain how users are scheduled.
- o describe the function of the Backstop process.
- o explain how a select group of operator commands relate to process management.
- o examine process management-related data structures in memory.

STATE DIAGRAM



PROCESS EXCHANGE

Process Exchange is the hardware/firmware mechanism used to switch the CP from being used by one user to being used by a different user.

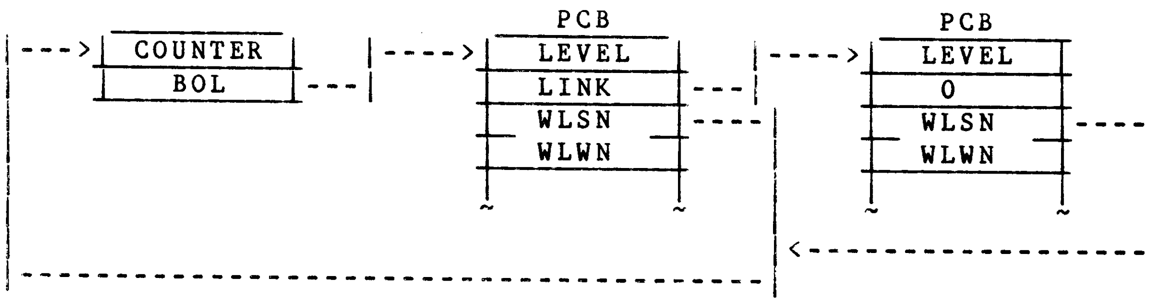
A context switch occurs whenever a higher priority user or system requires the use of the CP. The context switch involves saving the registers and state of the currently running process and placing the needed information in the current register set for the new user or system. This is accomplished by the firmware/hardware and the multiple user register sets in the High Speed Register File.

A process is a sequential flow of execution (a user, an I/O driver). The process is described to PRIMOS by a PCB (Process Control Block). Each process has its own PCB. A process must be in one of two states:

1). waiting for an event or non-CP resource
2). ready to execute.

When the process has all the resources required to run and is only waiting for the CP, the process′ PCB is placed on the READY LIST. If the process is waiting, its PCB is threaded onto a semaphore or wait list.

WAIT LIST (Semaphore)



WAIT <semaphore name>

access semaphore
count = count + 1
if count > 0
    then PCB --> Wait List
else process continues

NOTIFY <semaphore name>

access semaphore
count = count - 1
first PCB --> Ready List

USE OF LOCK SEMAPHORES - Simple Lock

```
                   _______________
                  |               |
A -------------> |               | <------------ B
                  |     DATA      |
```

Two processes are sharing the same data area. Process A could be changing data at the same time as Process B is reading the data. B may read incorrect data.

To prevent this, use a Simple Lock Semaphore (initial count = -1).

In order to access the data
    Process A must wait on the semaphore (count = 0)
    Process A proceeds

If Process B attempts to access the data it must first wait on the semaphore. (count = 1)
    Process B goes onto the Wait List for that semaphore
    Process A must NOTIFY the semaphore. (count = 0)
    Process B returns to the Ready List and proceeds

All processes that access the data must first WAIT on the semaphore and NOTIFY the semaphore when access is completed.

USE OF LOCK SEMAPHORES - Ordered Locks

```
           |-----------| 
           | SEMAPHORE |
|--------->|  DATA 1   |<---------|
A          |-----------|          B
|                                 |
|          |-----------|          |
|          | SEMAPHORE |          |
|--------->|  DATA 2   |<---------|
           |-----------|
```

Two processes are sharing two data areas.
If using simple locks;

Process A WAIT on semaphore 1
Process B WAIT on semaphore 2
Process B WAIT on semaphore 1
Process A WAIT on semaphore 2

A "Deadly Embrace" situation will be the result.

To avoid the "Deadly Embrace", it is vital that all processes that share data areas order their locks. The WAITs on the various semaphores must occur in the same order for each process.

Process A WAIT on semaphore 1	Process B WAIT on semaphore 1
Process A WAIT on semaphore 2	Process B WAIT on semaphore 2
Process A NOTIFY semaphore 1	Process B NOTIFY semaphore 1
Process A NOTIFY semaphore 2	Process B NOTIFY semaphore 2

SYSTEM LOCKS

The locks listed on the following page (in priority order) are used to control concurrent access to data areas. These locks utilize two semaphores (or wait lists).

Each lock consists of the following data structure:

COUNTER	
POINTER	READER'S Semaphore
COUNTER	
POINTER	WRITER'S Semaphore
USAGE Counter	
PRIORITY	

SYSTEM LOCKS

The system locks are listed in priority order, from lowest to highest.

FSLOK	Global file system lock
UFDLOK	UFD lock
SDLOK	Segment directory locks
TRNLOK	Transaction locks
UTLOK	Unit tables lock
RATLOK	Record availability lock
DEVLCK	Device table in PBDIOS
SP1LCK	
NETLCK	Network data
NMMLCK	Network memory mapping lock
SLCLCK	Smlc driver data
MOVLCK	segment mover lock (MOVUTU)
SHRLCK	Shared segment data lock
SEGLCK	GETSEG/RTNSEG lock (Segment tables)
PAGLCK	Page tables LOCK

PROCESS CONTROL BLOCK

0	LEVEL (PRIORITY)
1	LINK
2	POINTER TO WAIT LIST
3	"
4	ABORT FLAGS
5	MULTISTREAM CONTROL
6	RESERVED
7	"
'10	PROCESS ELAPSED TIMER
'11	"
'12	DTAR 2
'13	"
'14	DTAR 3
'15	"
'16	PROCESS INTERVAL TIMER
'17	"
'20	REGISTER SAVE MASK
'21	KEYS
'22	
..	REGISTER SAVE AREA
'61	
'62	RING 0 FAULT VECTOR
'63	"
'64	RING 1 FAULT VECTOR
'65	"
'66	NOT USED
'67	
'70	RING 3 FAULT VECTOR
'71	"
'72	PAGE FAULT VECTOR
'73	"
'74	CONCEALED STACK FIRST FRAME PTR
'75	CONCEALED STACK NEXT FRAME PTR
'76	CONCEALED STACK LAST FRAME PTR
'77	RESERVED

PRIORITIES

LEVEL	
0	CLOCK PROCESS/FNTSTOP
1	ASYNC. CONTROLLER PROCESSES
2	SYNC. CONTROLLER PROCESSES
3	MPC PROCESS, MP2
4	VERSATEC PROCESS, MPC-4
6	RING NET CONTROLLER PROCESS
7	DISK, ROIPQNM PROCESSES
N	NETMAN
8	SUPERVISOR PROCESS
9	USER LEVEL 3
10	USER LEVEL 2
11	USER LEVEL 1 (DEFAULT LEVEL)
12	USER LEVEL 0
13	IDLE
14	SUSPEND
15	BK1PCB (BACKSTOP 1) CPU #1
	BK2PCB (BACKSTOP 2) CPU #2
16	END OF READY LIST = 1

READY LIST EXAMPLE #1

```
PPA  | LEVEL A | PCB A |            PPB  | LEVEL B | PCB B |

'600 | BOL 0   |
'601 | EOL 0   |
'602 | BOL 1   |
'603 | EOL 1   |
'604 | BOL 2   |
'605 | EOL 2   |
'606 | BOL 3   |
'607 | EOL 3   |
                    PCB
'614 | BOL 7   |   | Level |
'615 | EOL 7   |   |   0   |

'624 | BOL 10  |
'625 | EOL 10  |    PCB            PCB            PCB
'626 | BOL 11  |   | Level |      | Level |      | Level |
'627 | EOL 11  |   | Link  |      | Link  |      |   0   |
'630 | BOL 12  |
'631 | EOL 12  |
                    PCB            PCB
'636 | BK1PCB  |   | Level |      | Level |
'637 | BK2PCB  |   | Link  |      |   0   |
'640 |    1    |
```

To move a PCB from the Ready List to a Wait List, the WAIT instruction is used. The NOTIFY instruction will move a process from a wait list to the Ready List. Both instructions must always reference a semaphore or wait list. The NOTIFY removes the first PCB from the semaphore and places it onto the Ready List at the proper level. When the process has completed execution or requires another resource, a WAIT is executed and the process moves from the Ready List to the specified Wait List or semaphore. PCBs are placed in the Wait List queue in priority level order.

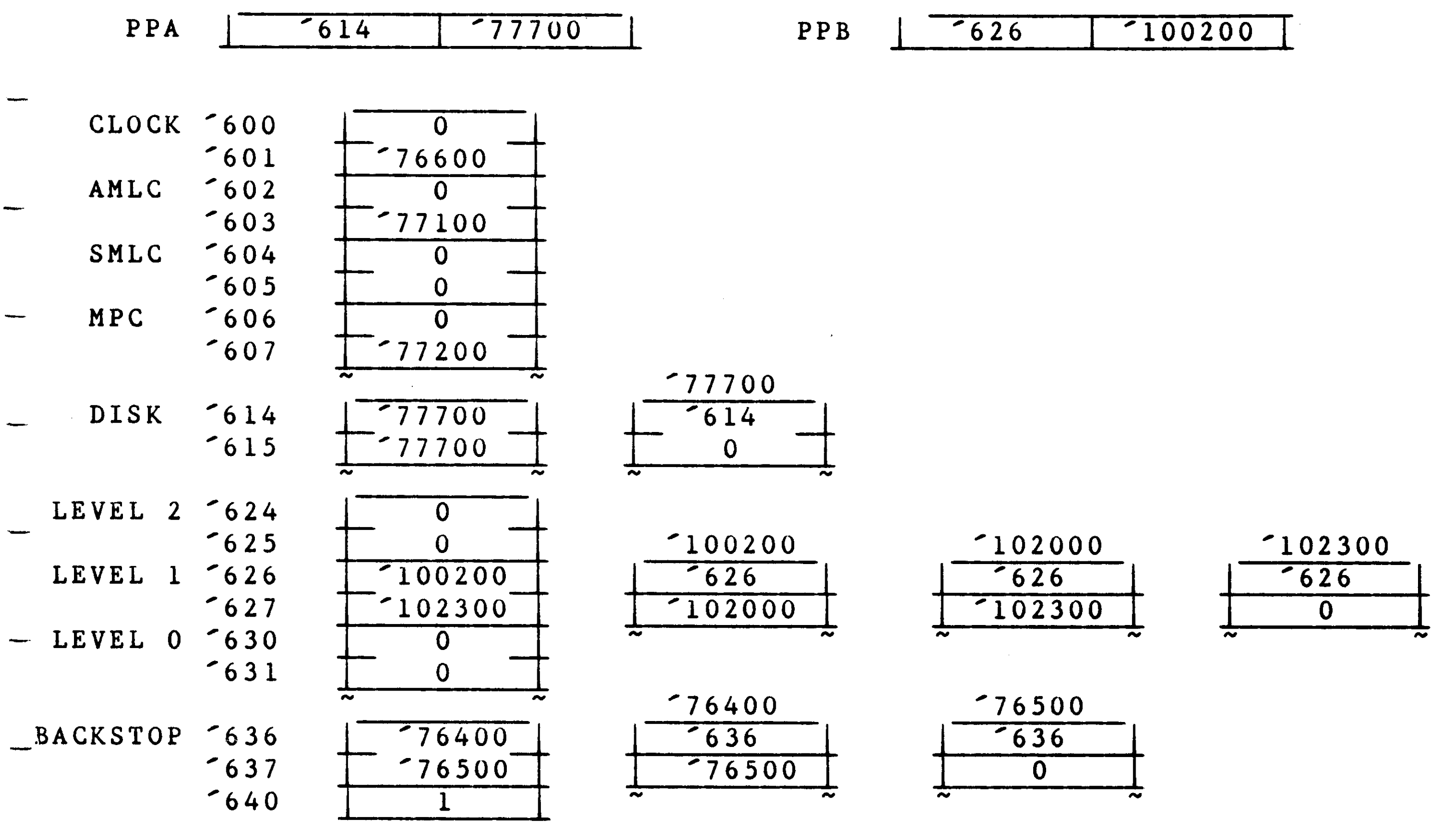
READY LIST

The firmware dispatcher uses two locations in the High Speed Register File Group 0. The first location is called PPA . PPA holds the pointer to the PCB of the currently running process. PLA contains the Ready List level of the currently running process. The currently running process will be the highest priority process on the Ready List. PPB contains the PCB address of the next process to run. PLB has the level of the next process. This allows the User Register Set for the next process to be set up while still running another process at a higher level.

The Ready List and the PCBs are all in Segment 4. This is one of the 'wired' segements of PRIMOS. This means it never gets paged out to the paging disk. The Ready List begins at Segment 4, address '600 and extends through address '640.

The PCB address and User Number bear a direct relationship to one another. For example; the address for User 1's PCB is 100100. The address for User 7's PCB is 100700. The PCB at address 101200 belongs to User 10. Addresses are in octal, user numbers are decimal. All PCBs are 64 ('100) words long so the least significant two octal digits of any PCB address is '00.

READY LIST EXAMPLE #2



This example shows actual addresses found using VPSD.
The contents pf PPA/PPB are calculated.

READY LIST EXAMPLE #3

PPA	'600	'76600
PPB	'614	'77700

SEGMENT #4

CLOCK	'600	'76600
	'601	'76600
AMLC	'602	0
	'603	'77100
SMLC	'604	0
	'605	0
MPC	'606	0
	'607	'77200
DISK	'614	'77700
	'615	'77700
LEVEL 2	'624	0
	'625	0
LEVEL 1	'626	'100200
	'627	'102300
LEVEL 0	'630	0
	'631	0
BACKSTOP	'636	'76400
	'637	'76500
	'640	1

'76600
'600
0

'77700
'614
0

'100200	'102000	'102300
'626	'626	'626
'102000	'102300	0

'76400	'76500
'636	'636
'76500	0

READY LIST EXAMPLE #4

PPA	'614	'77700
PPB	'626	'100200

SEGMENT #4

	Address	Contents
CLOCK	'600	0
	'601	'76600
AMLC	'602	0
	'603	'77100
SMLC	'604	0
	'605	0
MPC	'606	0
	'607	'77200
DISK	'614	'77700
	'615	'77700
LEVEL 2	'624	0
	'625	0
LEVEL 1	'626	'100200
	'627	'102300
LEVEL 0	'630	0
	'631	0
BACKSTOP	'636	'76400
	'637	'76500
	'640	1

'77700

'614
0

'100200

'626
'102000

'102000

'626
'102300

'102300

'626
0

'76400

'636
'76500

'76500

'636
0

READY LIST EXAMPLE #5

PPA	'626	'100200
PPB	'626	'102000

SEGMENT #4

CLOCK	'600	0
	'601	'76600
AMLC	'602	0
	'603	'77100
SMLC	'604	0
	'605	0
MPC	'606	0
	'607	'77200
DISK	'614	0
	'615	'77700
LEVEL 2	'624	0
	'625	0
LEVEL 1	'626	'100200
	'627	'102300
LEVEL 0	'630	0
	'631	0
BACKSTOP	'636	'76400
	'637	'76500
	'640	1

'100200	'102000	'102300
'626	'626	'626
'102000	'102300	0

'76400	'76500
'636	'636
'76500	0

READY LIST EXAMPLE #6

PPA	'626	'102000
PPB	'626	'102300

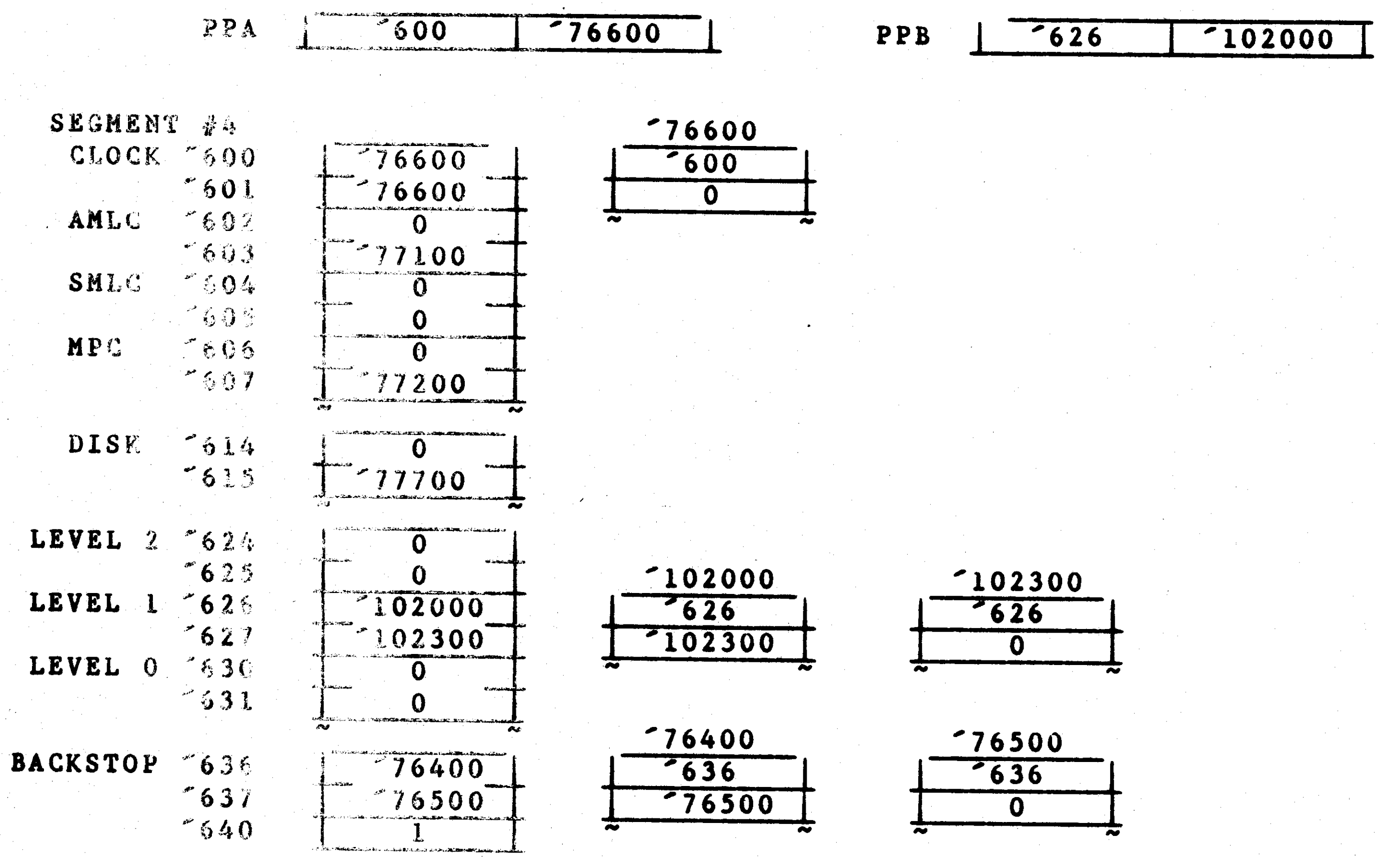
SEGMENT #4

CLOCK	'600	0
	'601	'76600
AMLC	'602	0
	'603	'77100
SMLC	'604	0
	'605	0
MPC	'606	0
	'607	'77200
DISK	'614	0
	'615	'77700
LEVEL 2	'624	0
	'625	0
LEVEL 1	'626	'102000
	'627	'102300
LEVEL 0	'630	0
	'631	0
BACKSTOP	'636	'76400
	'637	'76500
	'640	1

'102000	'102300
'626	'626
'102300	0

'76400	'76500
'636	'636
'76500	0

READY LIST EXAMPLE #7



CLOCK TICK
1A WAITING

READY LIST EXAMPLE #8

PPA	'600	'76600
PPB	'614	'77700

SEGMENT #4

CLOCK	'600	'76600
	'601	'76600
AMLC	'602	0
	'603	'77100
SMLC	'604	0
	'605	0
MPC	'606	0
	'607	'77200
DISK	'614	'77700
	'615	'77700
LEVEL 2	'624	0
	'625	0
LEVEL 1	'626	'102000
	'627	'102300
LEVEL 0	'630	0
	'631	0
BACKSTOP	'636	'76400
	'637	'76500
	'640	1

'76600
'600
0

'77700
'614
0

'102000	'102300
'626	'626
'102300	0

'76400	'76500
'636	'636
'76500	0

READY LIST EXAMPLE #9

PPA	´614	´77700
PPB	´626	´102000

SEGMENT #4

CLOCK	´600	0
	´601	´76600
AMLC	´602	0
	´603	´77100
SMLC	´604	0
	´605	0
MPC	´606	0
	´607	´77200
DISK	´614	´77700
	´615	´77700
LEVEL 2	´624	0
	´625	0
LEVEL 1	´626	´102000
	´627	´102300
LEVEL 0	´630	0
	´631	0
BACKSTOP	´636	´76400
	´637	´76500
	´640	1

´77700
´614
0

´102000	´102300
´626	´626
´102000	0

´76400	´76500
´636	´636
´76500	0

READY LIST EXAMPLE #10

PPA	'626	'102000
PPB	'626	'102300

SEGMENT #4

CLOCK	'600	0
	'601	'76600
AMLC	'602	0
	'603	'77100
SMLC	'604	0
	'605	0
MPC	'606	0
	'607	'77200
DISK	'614	0
	'615	'77700
LEVEL 2	'624	0
	'625	0
LEVEL 1	'626	'102000
	'627	'102300
LEVEL 0	'630	0
	'631	0
BACKSTOP	'636	'76400
	'637	'76500
	'640	1

'102000	'102300
'626	'626
'102300	0

'76400	'76500
'636	'636
'76500	0

READY LIST EXAMPLE #11

PPA		
PPA	'626	'102300

PPB		
PPB	'636	'76400

SEGMENT #4

CLOCK	'600	0
	'601	'76600
AMLC	'602	0
	'603	'77100
SMLC	'604	0
	'605	0
MPC	'606	0
	'607	'77200
DISK	'614	0
	'615	'77700
LEVEL 2	'624	0
	'625	0
LEVEL 1	'626	'102300
	'627	'102300
LEVEL 0	'630	0
	'631	0
BACKSTOP	'636	'76400
	'637	'76500
	'640	1

'102300

'626
0

'76400

'636
'76500

'76500

'636
0

READY LIST EXAMPLE #12

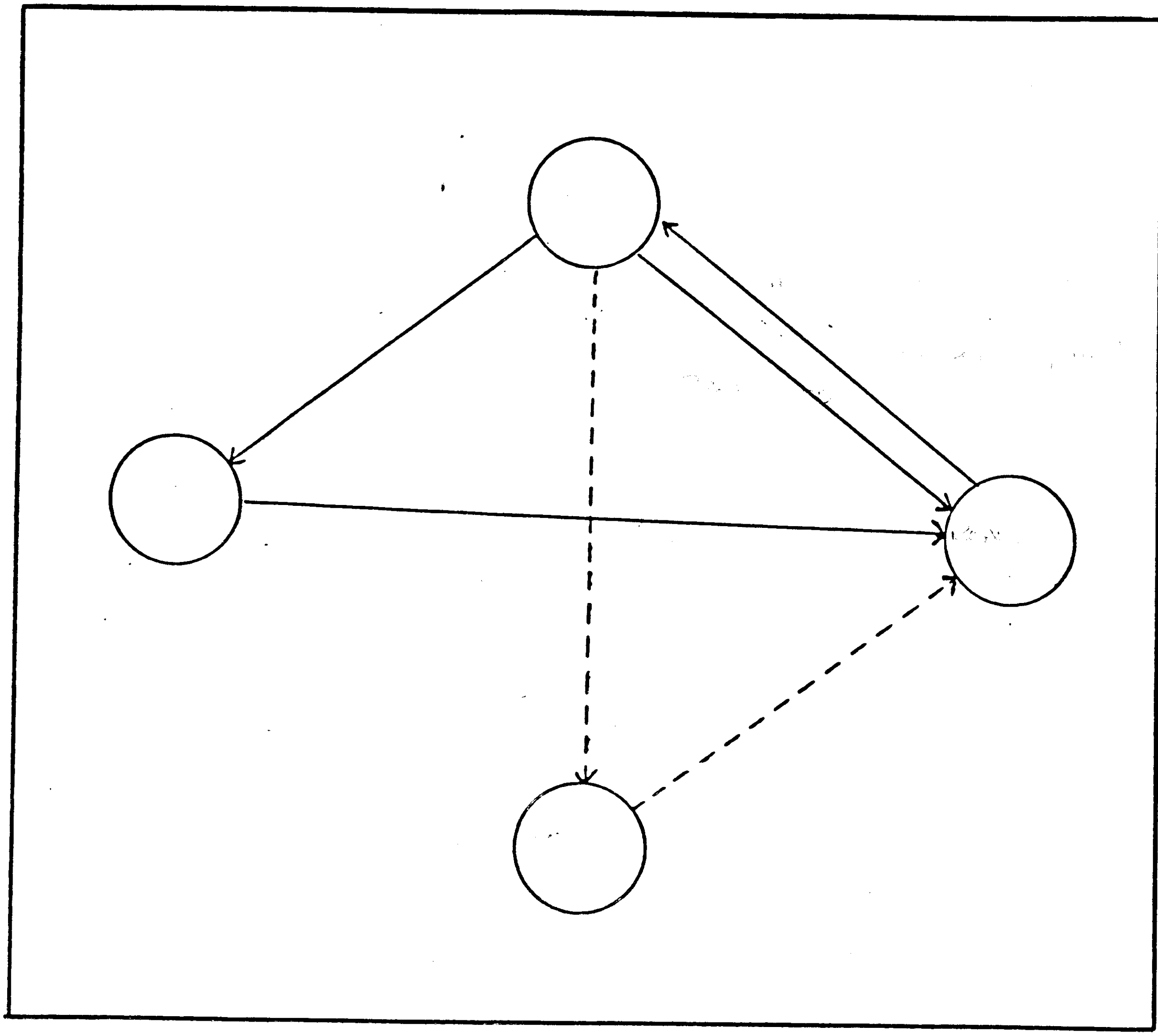
PPA	'636	'76400
PPB	'636	0

SEGMENT #4

CLOCK	'600	0
	'601	'76600
AMLC	'602	0
	'603	'77100
SMLC	'604	0
	'605	0
MPC	'606	0
	'607	'77200
DISK	'614	0
	'615	'77700
LEVEL 2	'624	0
	'625	0
LEVEL 1	'626	0
	'627	'102300
LEVEL 0	'630	0
	'631	0
BACKSTOP	'636	'76400
	'637	'76500
	'640	1

'76400	'76500
'636	'636
'76500	0

STATE DIAGRAM



SCHEDULING OF USERS

PRIMOS scheduling is based on two criteria.

1). PROCESS EXCHANGE
2). BACKSTOP PROCESS (SCHED)

The process exchange mechanism is implemented in firmware and uses the ready list/wait list philosophy described earlier.

SCHED, also known as the backstop process:

1). Responding to requests for users to be placed on one of three queues and allocating a time-slice.
2). Deciding the sequence of processes placed on the READY LIST.

SCHED maintains nine basic queues using semaphores.

1). High priority (HIPRIQ)
2). Eligibility (ELIGQ)
3-7). Low priority (LOPRIQ)
    3). Supervisor
    4). User level 3
    5). User level 2
    6). User level 1
    7). User level 0
8. Idle (IDLEQ)
9. Suspend (SUSPQ)

When a user process returns to command level, the listener is called to a invoke a new command level and CL$GET is called to read in the command line. C1IN$ is then called to read in the characters. C1IN$ will wait on BUFSEM (there is one BUFSEM semaphore per terminal user) and when a character is input into the user ring buffer the AMLC driver will notify BUFSEM. The user will continue to use C1IN$ to input characters until a <CR> character is detected.

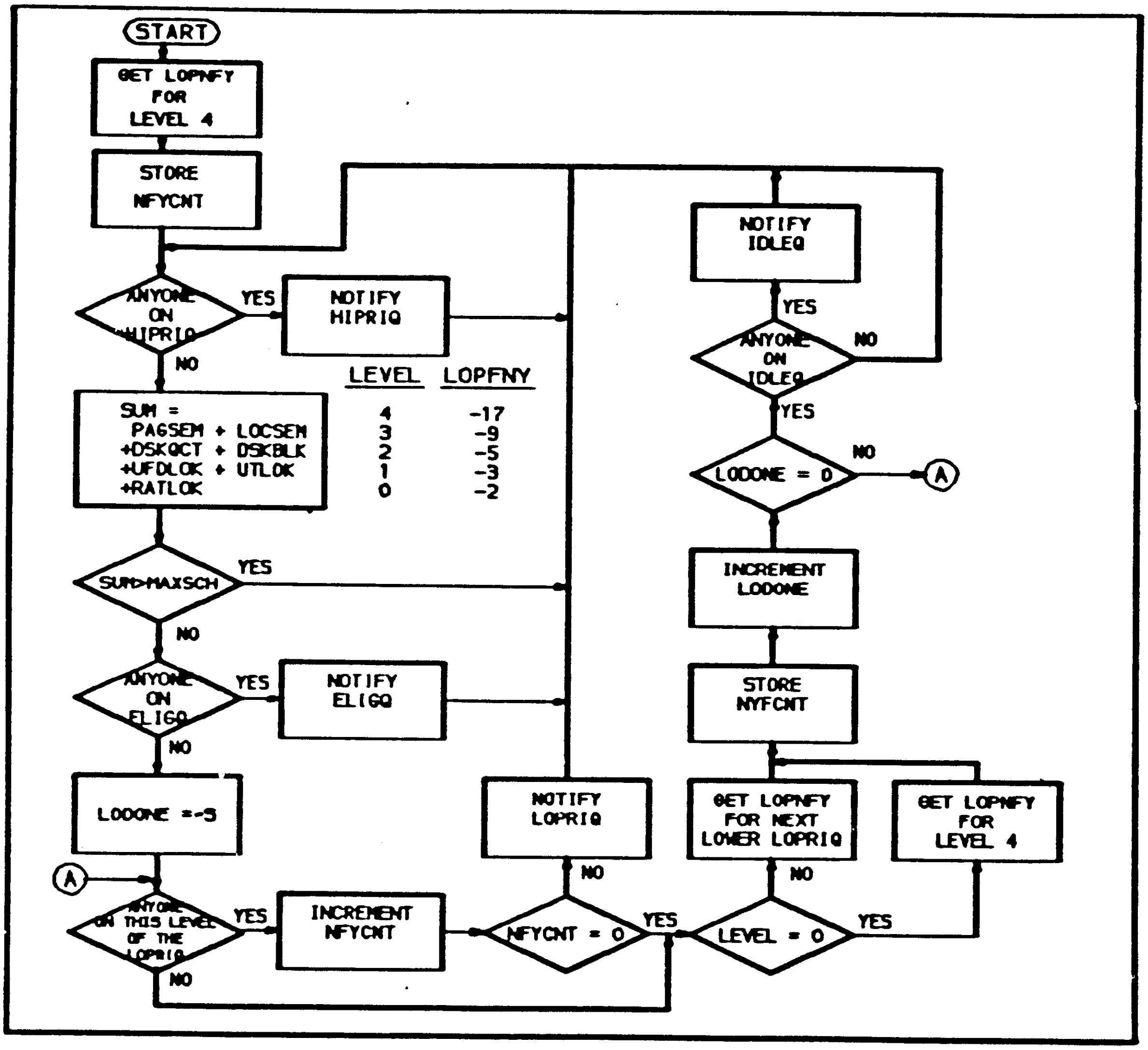
On detecting <CR> CL$GET calls SCHED to place the user process on the HIGH priority queue and to allocate a full time-slice. SCHED scans for high priority users before any others and a user in the high priority queue will be placed on the ready list and scheduled to run with a timeslice of 3/10 sec. At the end of this period the process will fault and be placed on the elgibility queue. The backstop process scans the elgibility queue after the high priority queue and eventually the user will be notified and moved on to the ready list with another timeslice of 3/10 sec.

SCEDULING OF USERS (CONT'D)

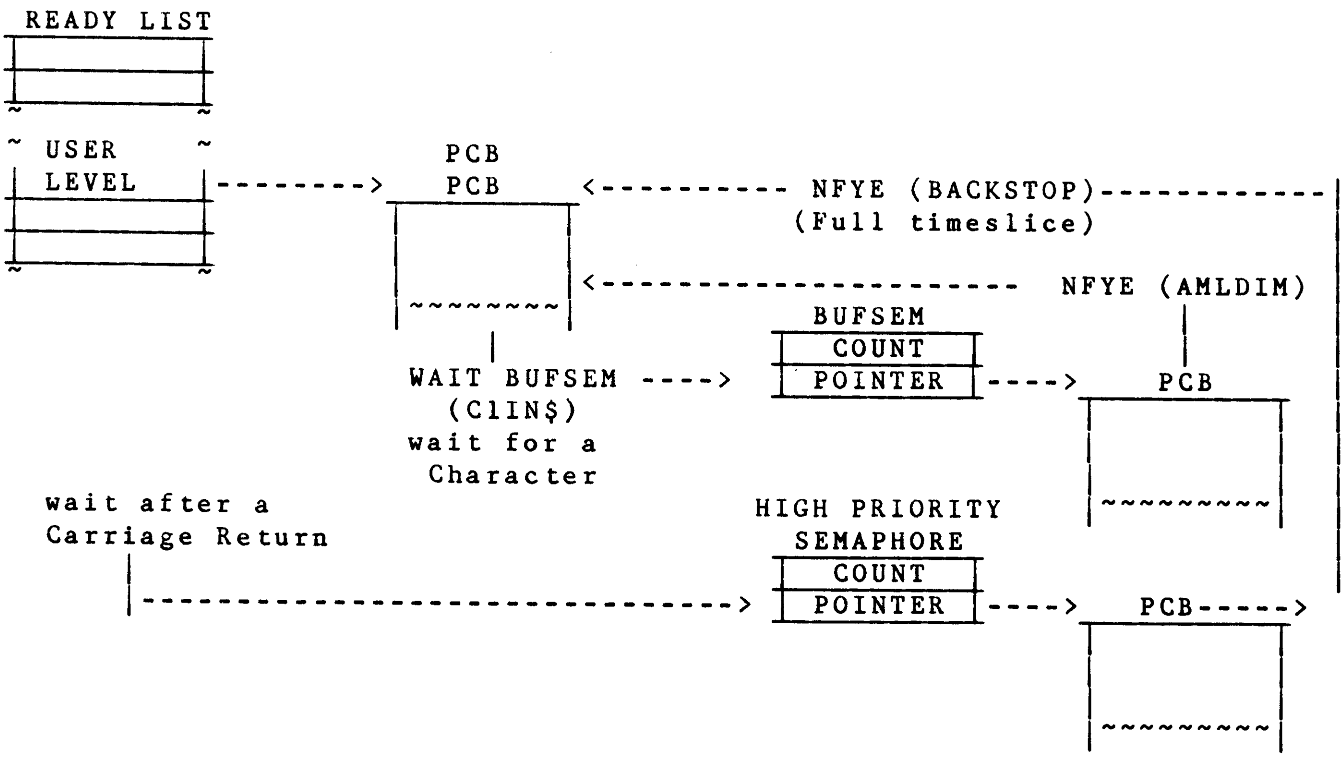
This sequence of events continues until the full 2 second time-slice has elapsed. The process is then placed on the low priority queue appropriate to its priority level, and is given a new 2 second timeslice.

The backstop process will schedule users on the low priority queues after both the high priority and the elgibility queues have been exhausted. The Idle level is checked only when there is no activity on the High Priority queue, the Eligibility queue, and all of the Low Priority queues.

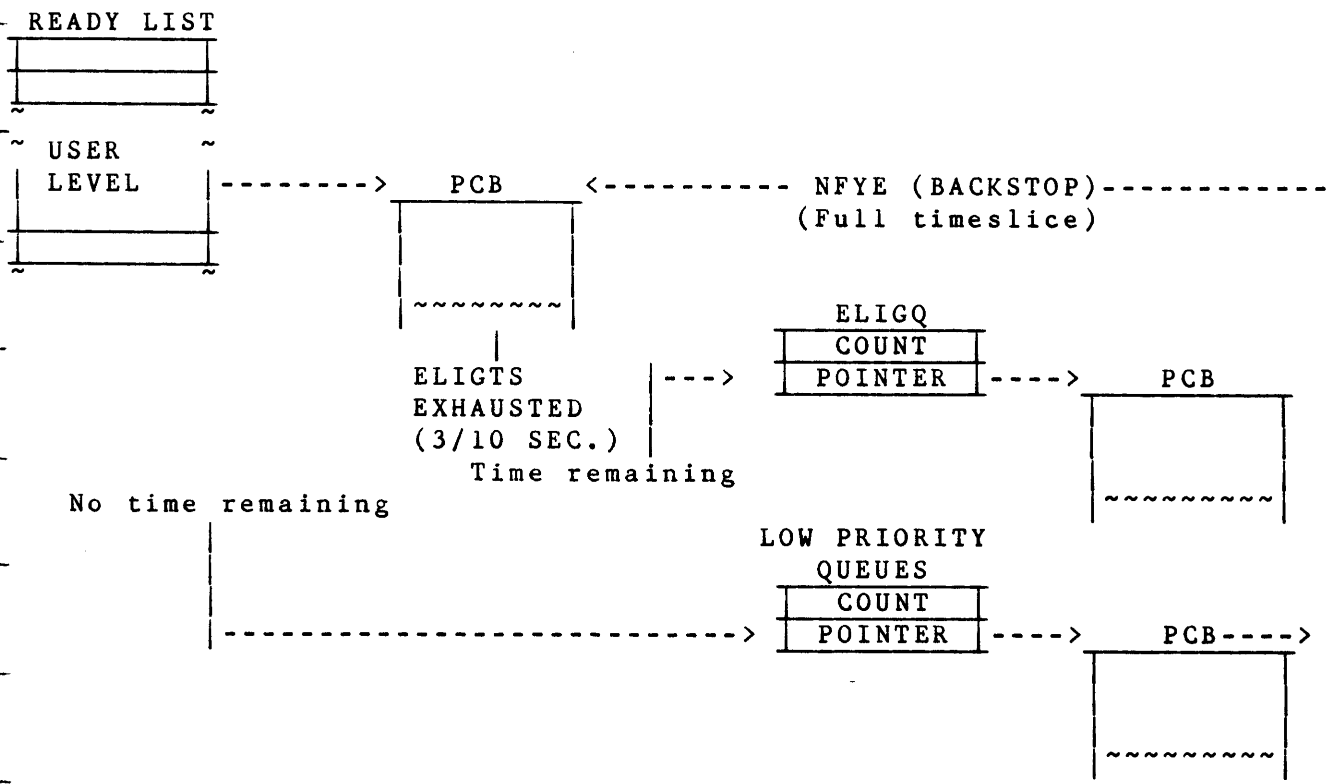
BACKSTOP PROCESS



INTERACTIVE USER



COMPUTE BOUND USER



USER PRIORITIES AND TIME-SLICE

The following operator command is available for changing user priorities and time-slice.

```
CHAP  [-USERNO/ALL] [PRIORITY] [TIME-SLICE]
                    [-IDLE]
                    [-SUSPEND]
```

USERNO	Is in the form -nn or ALL
PRIORITY	Integer 0 to 3 (default = 1)
TIME-SLICE	Length of time-slice in tenths of seconds. 0 means reset to the system default (2 sec.) If omitted the time-slice is unchanged.
-IDLE	Put process(es) into the IDLE state.
-SUSPEND	Put process(es) into the SUSPEND state.

If both priority and timeslice are omitted, then priority and time-slice are set to the system default values.

The following user command is available for changing user priorities and time-slice.

```
CHAP  [UP]
      [DOWN]
      [DEFAULT]
      [LOWER nnn] [timeslice]
   ** [IDLE]
```

** Can only be issued from a phantom

STAT US Displays the priority of users not at user level 1.

LOGOUT Resets priority and timeslice to defaults.

ELIGTS Is used to modify the eligibility time-slice from the system console. This will affect all users equally.

ELIGTS [<eligibility_timeslice>]   (default = 3/10 sec.)

MAXSCH

Previously, MAXSCH was determined by indexing into an array of values; 0,0,1,2,3,4,4. The value of the index was the memory size in 32K units. If there was more than 256K then MAXSCH would be 4.

MAXSCH is now calculated as follows:

MAXSCH = (megabytes_of_memory + 3) * x + y

where, x is 1.2 if there exists an alternate device on a different controller than the primary device, otherwise it is 1.
y is 1 if CPU is a P850, otherwise it is 0.

The optimal value of MAXSCH is application dependent, hence there is no hard and fast formula to determine its value. Therfore, it is a configurable parameter.

rule of thumb:

$$\text{MAXSCH} = \frac{\text{Physical-Memory-Size} - \text{PRIMOS-locked-memory}}{\text{average-job-size}}$$

Section 4 - Device Management

Objectives: The student will be able to:

- o describe how a DMx transfer occurs.
- o explain how the four types of DMx differ.
- o list I/O controllers and DMx methods used.
- o define an external interrupt.
- o describe how external interrupts are serviced.
- o describe how a clock interrupt is processed.
- o explain how terminal I/O is processed.
- o explain the allocation of terminal buffers.
- o explain how disk requests are serviced.
- o examine device management-related structures in memory with VPSD.
- o answer device management-related questions by examination of source code.

DMx Operation

DMx is a method whereby an I/O data/memory transfer may occur without software intervention. To perform such operations a temporary diversion in the sequence of microcode from CPU instruction to DMx transfer routines occurs. This is called cycle stealing or a TRAP. At the end of the DMx/memory transfer, the CPU instruction microcode continues as though nothing had happened. The actual trap diversion occurs at the end of the micro step in which it was sensed. At the same time, information about the next CPU micro step is saved to effect a return to the original sequence.

There are four types of DMx transfer: DMA, DMC, DMT, and DMQ. Each method has advantages and disadvantages in terms of speed, volume, and control features and so form a comprehensive range of methods.

DMA TRANSFERS

Used by Disk controllers, some Tape controllers, and PNC controllers.

0) Driver acquires channels at coldstart, and for each DMA transfer, performs the following setup operations:
   - o preload IOTLB,
   - o initialize channel with transfer address and range,
   - o output channel address to controller, and
   - o initiate read/write operation on device.

1) When ready to transfer data, the controller raises DMx request.

2) CPU scans the backplane for any DMx requests at the end of each microcode step. If there are pending requests, the CPU traps into the DMx microcode.

3) DMx microcode checks the backplane priority network and enables the DMx request from the highest priority controller. DMx microcode turns off the DMx request signal.

4) Controller places channel address onto the address bus and, over the control bus, indicates both the transfer direction and the type of DMx operation.

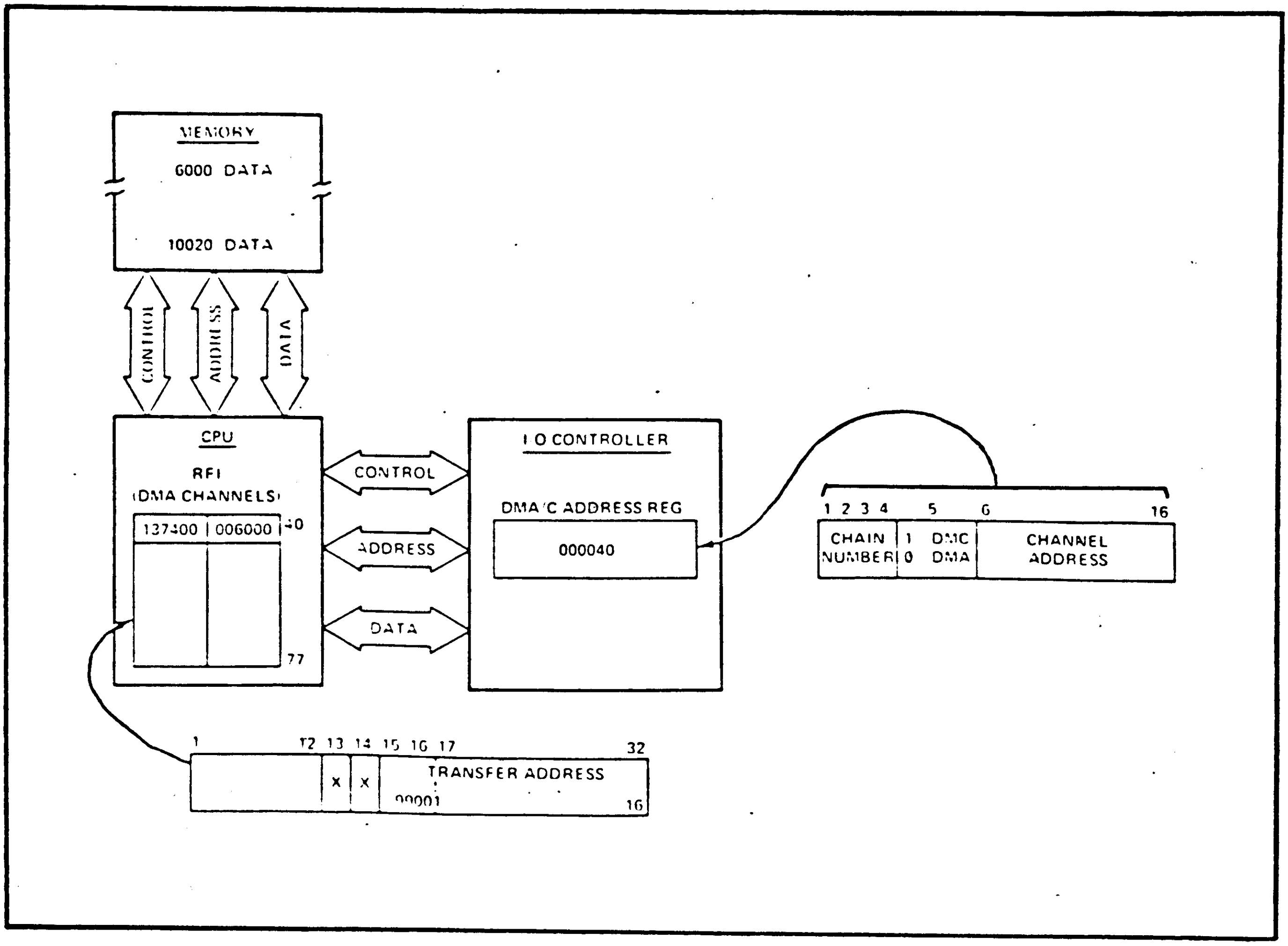
5) Upon receiving the above information, DMx microcode will
   - o transfer 16 bits of data,
   - o adjust transfer address and range, and
   - o check for EOR condition.

   If EOR, DMx microcode sends and EOR signal back to the controller.

6) DMx microcode checks for more pending DMx requests. If there are pending requests, go back to (3); if no pending requests, return to pre-DMx state.

7) Controller generates an EOR external interrupt upon receipt of EOR signal from DMx microcode.

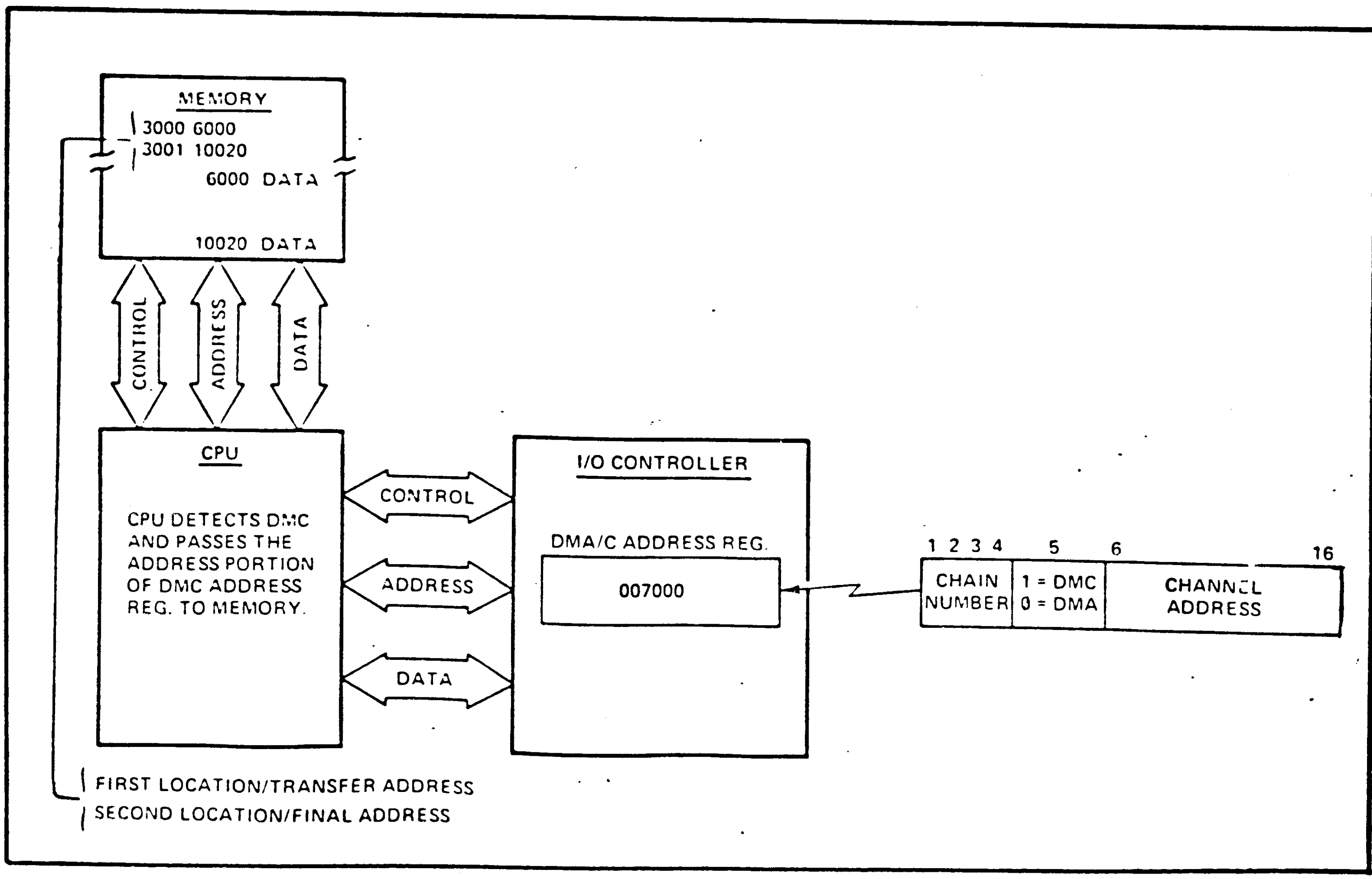
DMA TRANSFERS (CONT'D)



DMC TRANSFERS

Used by MDLC, SMLC, both AMLC and QAMLC for input, some tape controllers, and MPC controllers (hi-speed parallel printers).

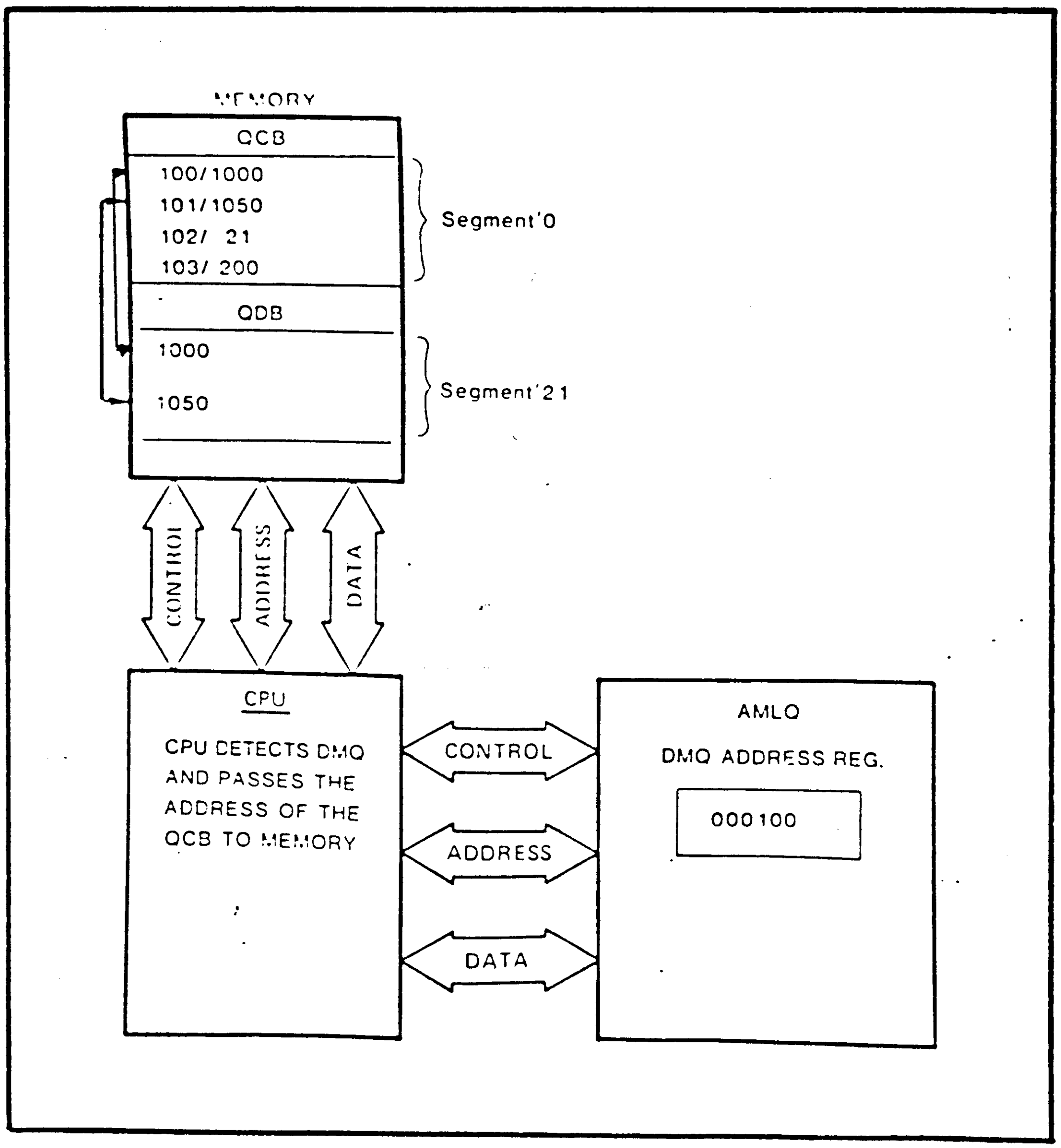
DMC uses pairs of memory locations in segment 0 to hold the starting and ending transfer addresses, respectively. Each pair is acquired by the driver at coldstart. Hence, the address presented by the controller to the CPU is the address of the first word in the pair. There is no explicit range; rather, the range will be implicit from the starting and ending transfer addresses.



DMQ TRANSFERS

Used by QAMLC for output, ICS1 and ICS2 for asynch (both input and output).

DMQ uses a QCB to hold the transfer control information. Each QCB is a four word data structure located in segment 0. The layout of the QCB is on the following page. Hence, the address presented by the controller to the CPU is the address of the QCB. The data buffer, the QDB, is NOT in segment 0. DMQ is the only form of DMx that allows the data to be outside of segment 0.



DMQ Operation

The control information is held in segment 0 of memory in an area known as the Queue Control Block (QCB).

Each queue is implemented by an array of 2**N words where N is greater than or equal to 4, and less than or equal to 16.

Each QCB is a four word structure:

```
    TOP POINTER (read)          word number of the head of the queue
    BOTTOM POINTER (write)      word number of the tail of the queue
    SEGMENT NUMBER or PHYSICAL ADDRESS
                                segment number or PPN of above pointers
    MASK                        2**N - 1 defines the size of the buffer
```

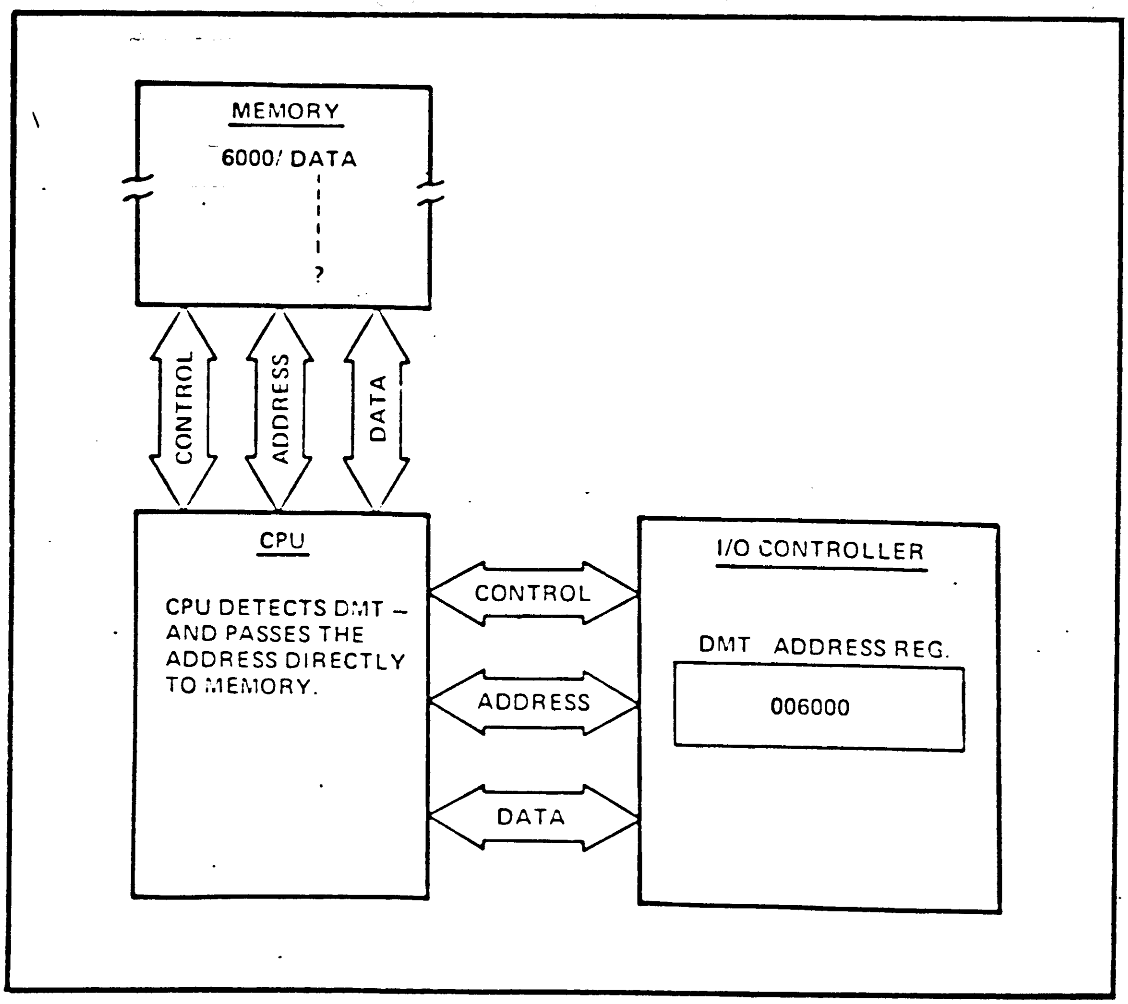
The instructions provided for DMQ and QUEUE manipulation are:

```
    ATQ                   : add to the top of the queue
    ABQ or DMQ input      : add to the bottom of the queue
    RTQ or DMQ output     : remove from top of the queue
    RBQ                   : remove from the bottom of the queue
    TSTQ                  : test the queue (# items->A, if empty EQ->CC)
```

DMT TRANSFERS

Used by disk controller for channel programs, AMLC for output, and downline loading ICSn microcode.

Unlike the other types of DMx, DMT does not place the responsibility for managing the transfer parameters upon the DMx microcode. Rather, the controller is responsible for updating the transfer address and the range.



EXTERNAL INTERRUPTS

How Interrupts Occur

(0) Interrupts must be enabled (bit 1 of the MODALS).

(1) Controller ships over the interrupt request to the CPU.

(2) CPU ´sees´ the request, but waits for the current instruction to complete.

(3) CPU disables interrupts (bit 1 of the MODALS).

(4) CPU ACKnowledges the controller.

(6) The controller,upon receiving the ACK, will ship its ´interrupt vector address´ to the CPU.

(7) CPU stores the current process´ PB (and P-Ctr) in PSWPB and its KEYS (and MODALS) in PSWKEYS (RFO).

(8) At this point, (software) control is transferred to segment 4, at the offset specified by the interrupt vector address.

PHANTOM INTERRUPT CODE

In order to NOTIFY a process, PIC must ensure that the PB and KEYS are restored before issuing the NOTIFY.

The PIC basically consists of one instruction, an INEC, with the name of a semaphore as the operand.

The INEC instruction performs the following actions:

1) Reload the PB and KEYS from PSWPB and PSWKEYS.
2) Issue a CAI to clean up the I/O bus.
3) Enable interrupts.
4) Notify the appropriate semaphore.

CLOCK PROCESS

The clock interrupt is treated like any other device interrupt. An address ('63) is presented to the CPU. The hardware interprets this location as the address of the Phantom Interrupt Code (PIC) in Segment 4 for this device. The PIC executes an INEC which acknowledges the interrupt, clears the Active Interrupt flag, and does a NOTIFY to CLKSEM.

The clock process will then be entered. Following is a general list of the functions performed:

1). Handle PBHIST.
2). Increment ONE-MINUTE timer.
    If zero, reset clock and set USER 1's MINALM abort flag and NOTIFY ASRSEM.
3). Increment timer 2 (Paper Tape Punch) (1/75 second).
    If zero, reset clock and call BRPDIM (if chars in buffer).
4). Increment Timer 3 (Digital input)
    If zero, reset timer and enter DIGDIM
5). Increment timer 4 (ASR) (1/30 or 1/10 second).
    If zero, reset clock and call ASRDIM.
6). Increment timer 5 (1/10 second).
    If zero , doing the following:
    A). Reset clock
    B). If sensor check has occurred,
        set USER 1's CHKALM Abort Flag
    C). Update clock ring
    D). Handle USER timer semaphores
    E). Increment Timer 9 (DISK) 1/2 second,
        If zero, reset clock and notify DSKSEM
    F). Increment Timer 10 (SMLC) 1/2 second,
        If zero, reset clock and set USER 1's SMLALM Abort Flag.
    G). Increment Timer 11 (Gross Network) 10 second,
        If zero, reset clock and notify PNTSEM (NETMAN).
    H). Increment Timer 12 (Network Protocol) 1 second,
        If zero, reset clock and notify PNTSEM.
    I). Increment Timer 13 (Remote USER I/O) 1/2 second,
        If zero, reset clock and notify PNTSEM.
    J). Increment Timer 14 (Date and Time) 4 second
        If zero, reset clock and update date and time for TIMNOW and DATNOW.
7). Increment Timer 15 (Real Time Queue) 1 second,
    If zero for any process, set process' TMOALM abort flag.
8). Handle timers for PNCDIM.
9). WAIT CLKSEM.

THE QAMLC DRIVER - AMLDIM

The QAMLC will configure itself to drive up to eight controllers using device addresses '54, '53, '52, '35, '15, '16, '17 and '32. The default configuration can be changed using the AMLC command at the system console or in PRIMOS.COMI

```
AMLC  [PROTOCOL]  LINE  [CONFIG]  [LWORD]
```

PROTOCOL

TTY	terminal protocol (default protocol)
TRAN	transparent protocol
TTYUPC	upper case output protocol
TTYNOP	ignore this line (used for assigned lines)
TT8BIT	8-bit protocol
ASD	auto-speed detect

LINE	The AMLC line number (octal)
CONFIG	See line configuration table.
LWORD	See LWORD table.

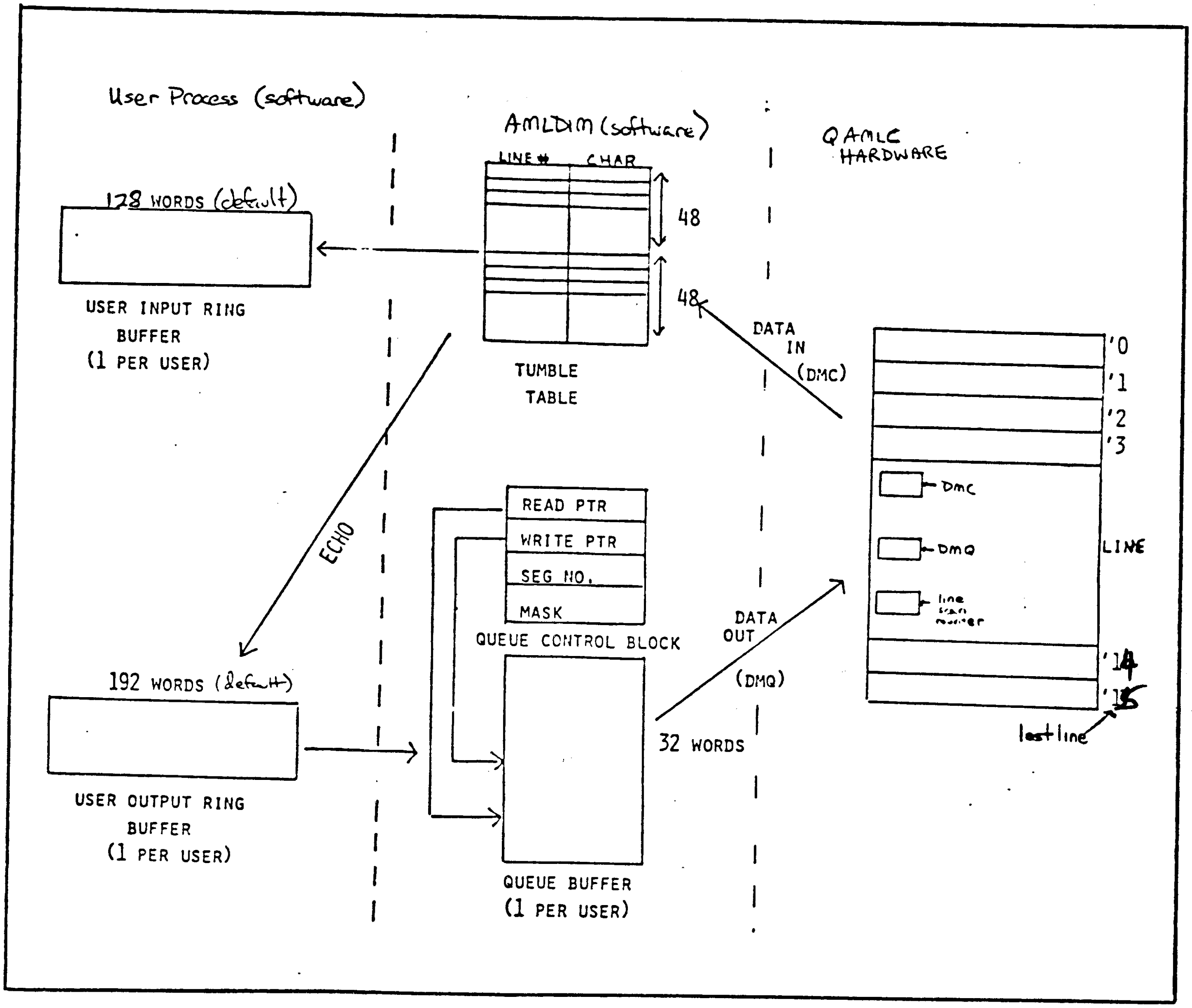
LINE CONFIGURATION TABLE

```
1  2  3  4   5   6   7   8  9   10   11   12   13   14   15  16
Line no.         |   |      |         |    |    |    |    Character
(bit 4 is lsb)   |   |      |         |    |    |    |     length
 set to 0        |   |      |         |    |    |    |       0 0 - 5 bits
                 |   |      |         |    |    |    |       1 0 - 6 bits
                 |   |      |         |    |    |    |       0 1 - 7 bits
      Data Set   |   |      |         |    |    |    |       1 1 - 8 bits
      control<-  |   |      |         |    |    |    |
  1 for modems       |      |         |    |    |    |-> Type of parity, 0 = odd
                     |      |         |    |    |
       loop line<-   |      |         |    |    |--> Parity disable, 0 = enable
     (for testing)          |         |    |             (default) 1 = disable
       Set to 0             |         |    |--->Stop bits
                            |         |          0 = 1 bit
                            |         |          1 = 2 bits
                            |         |
                            |         |----> Reverse Flow Control
                            |                0 = disabled (default)
                            |                1 = enabled (ICS1 only)
                            |
                       Line Speed
                   0 0 0 -  110 baud
                   0 0 1 -  134.5 baud
                   0 1 0 -  300 baud
                   0 1 1 -  1200 baud
                   1 0 0 -  program clock - default 9600 baud
                   1 0 1 -  75 baud
                   1 1 0 -  150 baud
                   1 1 1 -  1800 baud
```

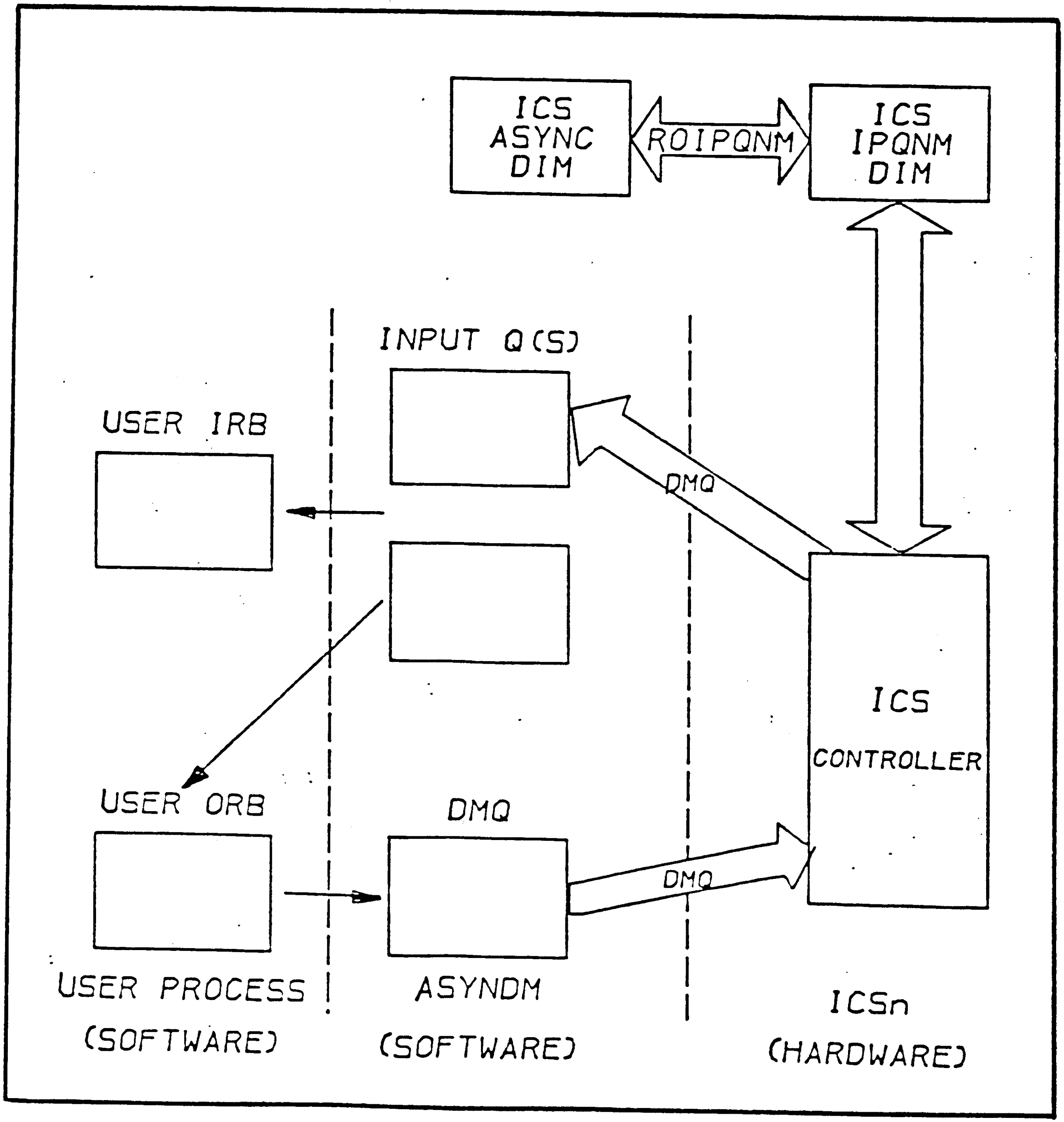
LWORD TABLE

```
1   2   3   4   5   6   7   8   9   10   11   12   13   14   15   16
|   |   |   |   |   |   |       ___________________________________
|   |   |   |   |   |   |                  USER NUMBER
|   |   |   |   |   |   |
|   |   |   |   |   |   |--> CHECK, Enable error detection
|   |   |   |   |   |           1 = Parity or IRB overflow
|   |   |   |   |   |        (send a NAK if parity or irb overflow sensed)
|   |   |   |   |   |
|   |   |   |   |   |---> DSS hi/low, toggle for bit 5
|   |   |   |   |
|   |   |   |   |------> DSS enable, Check carrier, simulate XON/XOFF
|   |   |   |            ("buffered" or "reverse channel" protocol)
|   |   |   |
|   |   |   1 = When XOFF or DSS enabled, flag to show XOFF
|   |   |
|   |   0 = no xon/xoff
|   |   1 = xon/xoff
|   |
|   0 = LF echoed for CR  (only if half duplex)
|   1 = LF not echoed for CR
|
0 = Full duplex
1 = Half duplex
```

QAMLC BLOCK DIAGRAM



ICS BLOCK DIAGRAM



LIOCOM

```
A(LIOCOM) ----->  -----------------------
                 |      ADDRESS OF       |
                 |   CONSOLE´S ORB       |
                  -----------------------
                 |      ADDRESS OF       |
                 |     USER 2´S ORB      |
                  -----------------------
                 |      ADDRESS OF       |
                 |     USER 3´S ORB      |
                  -----------------------
                 |      ADDRESS OF       |
                 |     USER 4´S ORB      |
                  -----------------------
                 ~                       ~
                 ~                       ~
                  -----------------------
                 |      ADDRESS OF       |
                 |   CONSOLE´S IRB       |
                  -----------------------
                 |      ADDRESS OF       |
                 |     USER 2´S IRB      |
                  -----------------------
                 |      ADDRESS OF       |
                 |     USER 3´S IRB      |
                  -----------------------
                 |      ADDRESS OF       |
                 |     USER 4´S IRB      |
                  -----------------------
                 ~                       ~
                 ~                       ~
```

A(ORB) = A(LIOCOM) + (2 * (BUFFER NUMBER + 1))

A(IRB) = A(LIOCOM) + (2 * (BUFFER NUMBER + 1)) +
(2 * NUMBER OF PROCESSES)

DISK I/O WAIT TIME

Disk I/O time = wait time + seek time + rotation time + transfer time

Wait time is the time a process must wait before its disk request is acted upon.

Wait (1) for a disk queue request block

(2) in a work list

DISK QUEUE REQUEST BLOCKS

FORWARD THREAD
SEMAPHORE
DEVICE TYPE
UNIT SELECT BITS
CYLINDER NUMBER
HEAD/RECORD NUMBERS
VIRTUAL BUFFER ADDRESSES

PHYSICAL PAGE ADDRESSES

NUMBER OF WORDS/CHANNEL
NUMBER OF WORDS/CHANNEL
NUMBER OF WORDS/CHANNEL
NUMBER OF CHANNELS
TOTAL TIME
ERROR MESSAGE INFO

7 queue request blocks at revision 18
17 queue request blocks at revision 19.1
32 queue request blocks at revision 19.3

DSKBLK is the semaphore processes must wait on to obtain a queue request block.

DISK I/O SEEK TIME

Disk I/O time = wait time + seek time + rotation time + transfer time

Seek time is the time a process must wait for the heads to move over the desired cylinder.

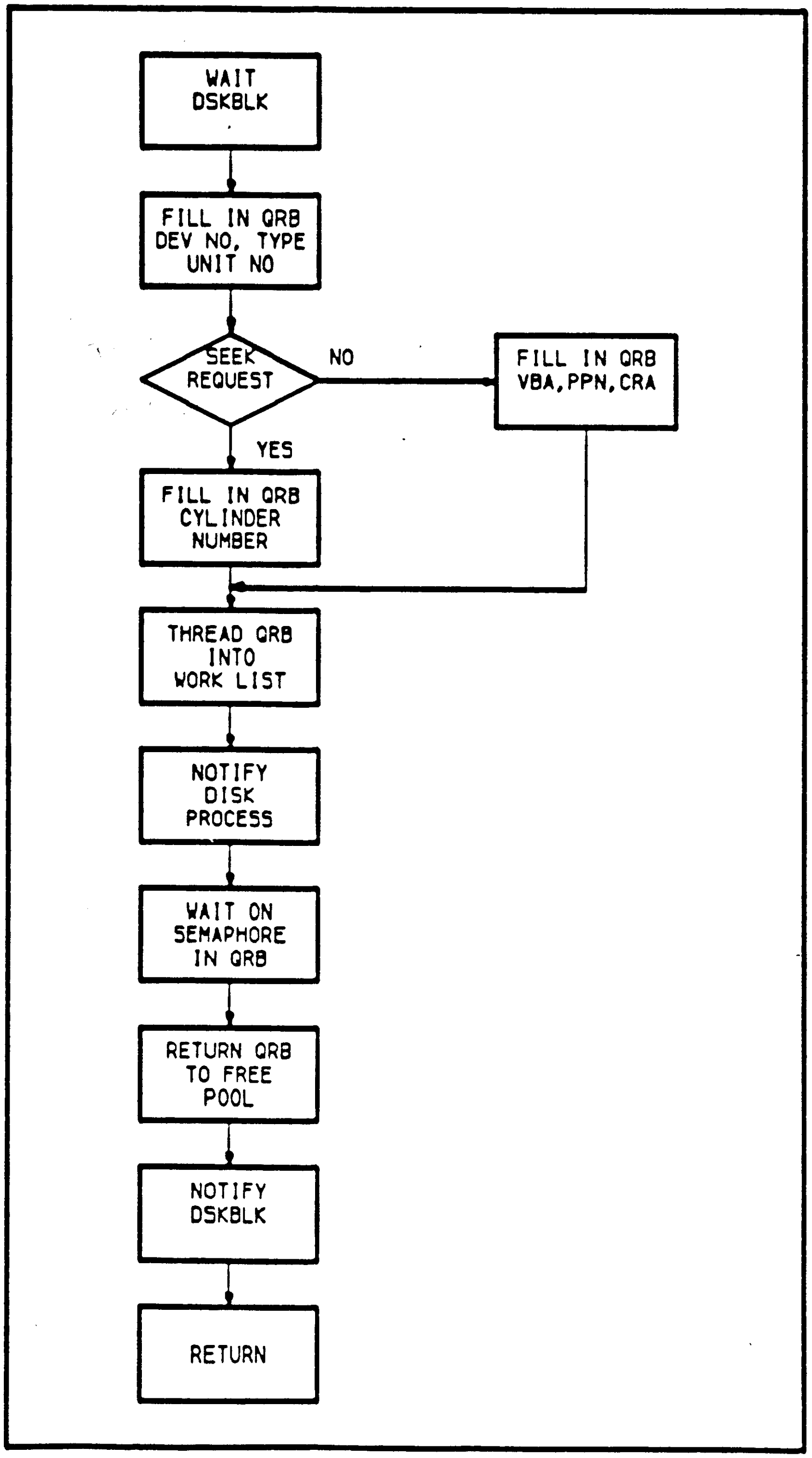
DISK I/O ROTATION AND TRANSFER TIMES

Disk I/O time = wait time + seek time + rotation time + transfer time

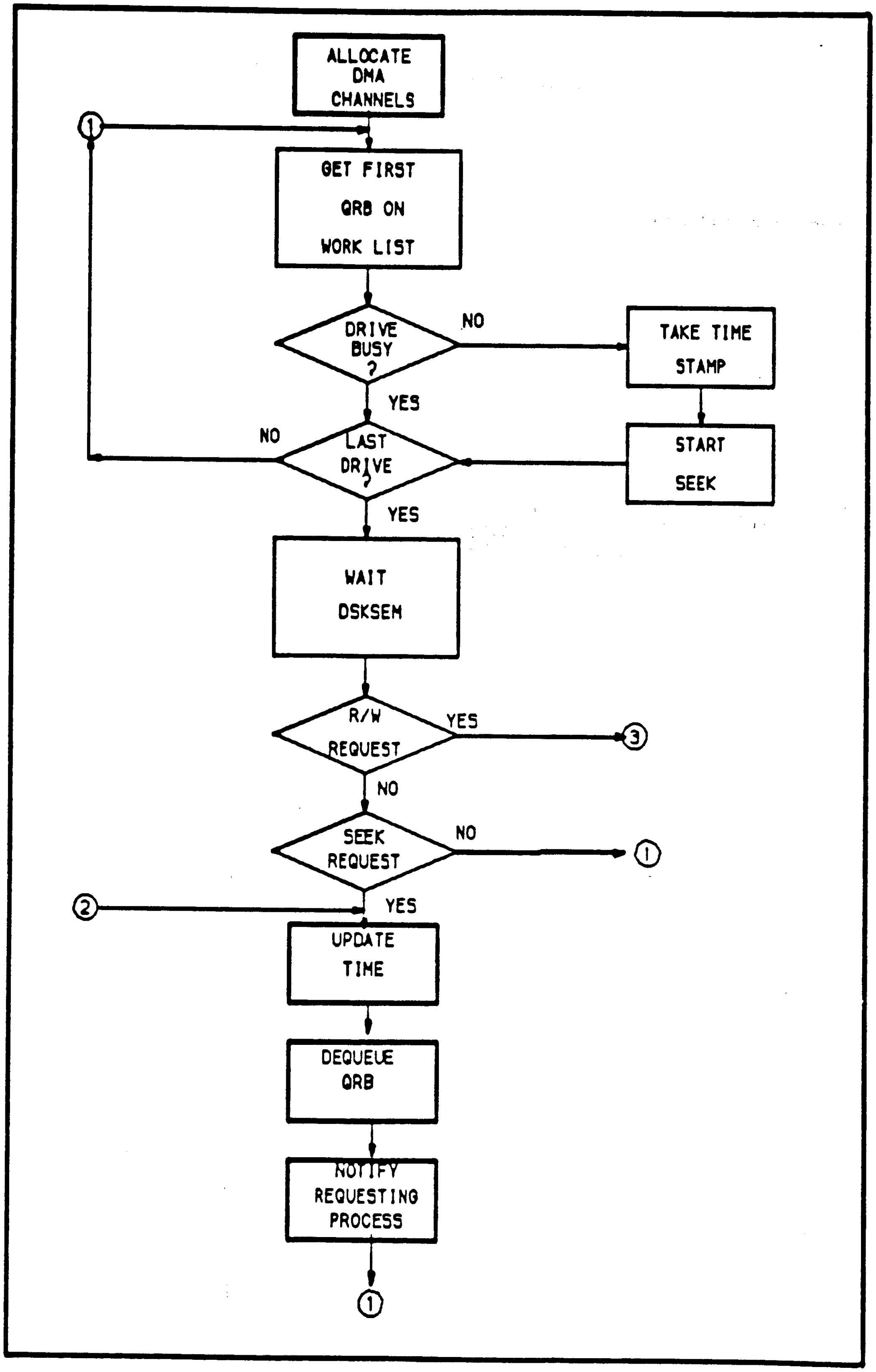
Rotation time is the time required for the drive to make one complete revolution.

Transfer time is the time required to do the actual physical data transfer.

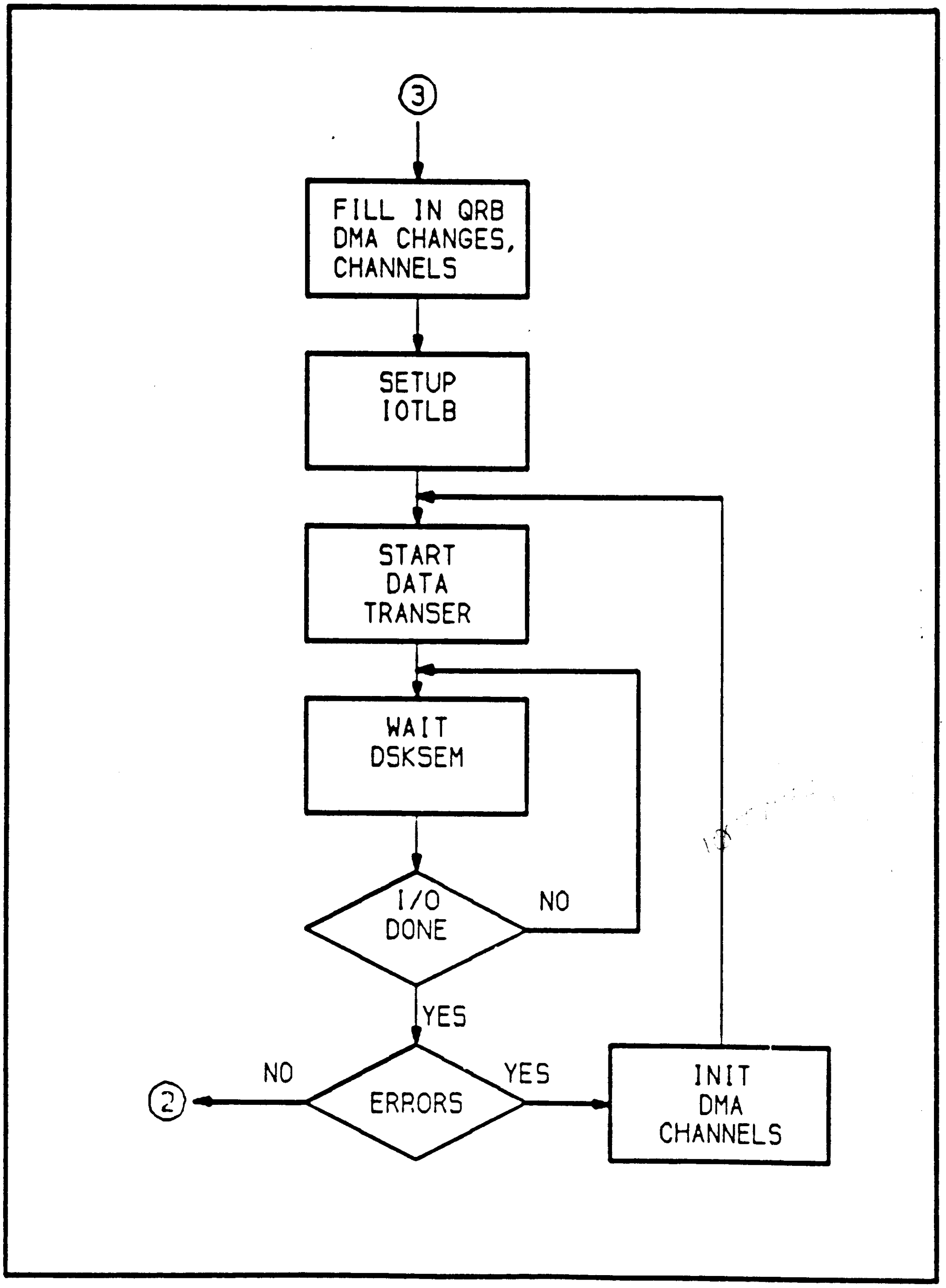
DISKIO.PMA - CALL SIDE



DISKIO.PMA - DISK PROCESS CODE



DISKIO.PMA - DISK PROCESS CODE (continued)



Section 5 - Procedure Management

Objectives: The student will be able to:

- o describe the contents of a user register set.
- o explain the use of the PB, LB, and SB registers.
- o describe the functions of the PCL mechanism.

THE USER REGISTER SET

	HIGH	LOW	
0			GR0
1			GR1
2	A	B	GR2
3	EH	EL	GR3
4			GR4
5	S/Y		GR5
6			GR6
7	X		GR7
10	FAR0		
11	FLR0		
12	FAR1/FAC		
13	FLR1/FAC		
14	PB		
15	SB		
16	LB		
17	XB		
20	DTAR3		
21	DTAR2		
22	DTAR1		
23	DTAR0		
24	KEYS/MODALS		
25	OWNER		
26	FCODE		
27	FADDR		
30	CPU TIMER		
31	MICROCODE SCRATCH		
.	"		
.	"		
37	"		

THE USER REGISTER SET CONTENTS

A	Accumulator Register	
B	Accumulator Extension (A + B = L)	
EH,EL	Accumulator Extension for long integers (64 bit)	
S	Stack Register (R S Modes)	
Y	Alternate Index Register (V Mode only)	
X	Index Register (R, S, V Modes)	
GR0-GR7	General Registers 0-7 (I Mode only)	
FAR0	Field Address Register 0	
FLR0	Field Length Register 0	
FAR1	Field Address Register 1	(for block moves
FLR1	Field Length Register 1	char./dec. data)
FAC	Floating Point Accumulator	
PB	Procedure Base Register	
SB	Stack Base Register	
LB	Link Base Register	
XB	Auxiliary Base Register	
OWNER	Address of User Register Set Owner´s PCB	
FCODE	Fault Code	
FADDR	Fault Address	
CPU TIMER	overflow of two´s complement value ends timeslice	

User programs may access the Register-file using LDLR and STLR (64V).
Only locations ´0 - ´17 are accessible.
Any attempt to access location ´14 (PB) will give undefined results.
The first eight locations are interpreted for V-mode (default).

PROCEDURE/LINK/STACK ARCHITECTURE

PROCEDURE AREA
- 1 per system if shared
- contains pure code and literals
- pointed to by Procedure Base Register (PB)

LINKAGE AREA
- 1 per user
- contains local variables and pointers
- pointed to by Linkage Base Register (LB)

STACK FRAME
- 1 per invocation
- contains caller's saved state, argument pointers, and dynamic work space
- pointed to by Stack Base Register (SB)

KEYS

bit #	purpose S R Modes		V I Modes
1	Arithmetic Error Cond.		C Bit
2	Double Precision Bit		reserved
3	reserved		Link
4-6	Mode bits		Mode Bits
	000	16S mode	
	001	32S	
	011	32R	
	010	64R	
	110	64V	
	100	32I	
7	reserved		Floating Point Exception
8	reserved		Integer Exception
9	Bits 9-16 are bits 9-16		LT (less than) bit
10	of address 6		EQ (equal) bit
11	"		DEX (decimal exception)
12	"		Ascii 8 bit
13	"		Floating Point Round
14	"		In CHECK bit (850 only)
15	"		I bit - In Dispatcher
16	"		S bit - Save Done

SUBROUTINE CALLS

(1) CALLING PROGRAM

CALL
- calls a subroutine
- generates PCL (procedure call)

PCL
- addresses an ECB through a link
- calculates the ring number
- allocates the stack frame
- intializes the state of the called procedure
- transfers the argument pointers

AP
- generates the argument pointers for the PCL
- follows the PCL instruction
- format

```
    AP    ARG, TAG
      where TAG modifier can be:
        - S     variable is an argument
        - SL    variable is the last argument
        - *S    the argument is an indirect address
        - *SL   the argument is an indirect and the last
```

THE CALLED SUBROUTINE

(2) THE SUBROUTINE

ARGT
- does the last step of the PCL instruction
- executed only if a fault occurs during argument pointer transfers
- must be present if the subroutine requires arguments

ECB
- generates an Entry Control Block (ECB) to define a procedure entry point
- resides in a link frame
- format

```
    LABEL    ECB   PFIRST,,ARGDISP, NARGS, SFSIZE,KEYS
       where
           PFIRST    - pointer to first executable statement
           ARGDISP   - displacement in the stack frame of the
                       argument list (default is '12)
           NARGS     - number of arguments to be passed
           SFSIZE    - stack frame size, the default is given
                       by the DYMN
           KEYS      - keys, the default is 64V
```

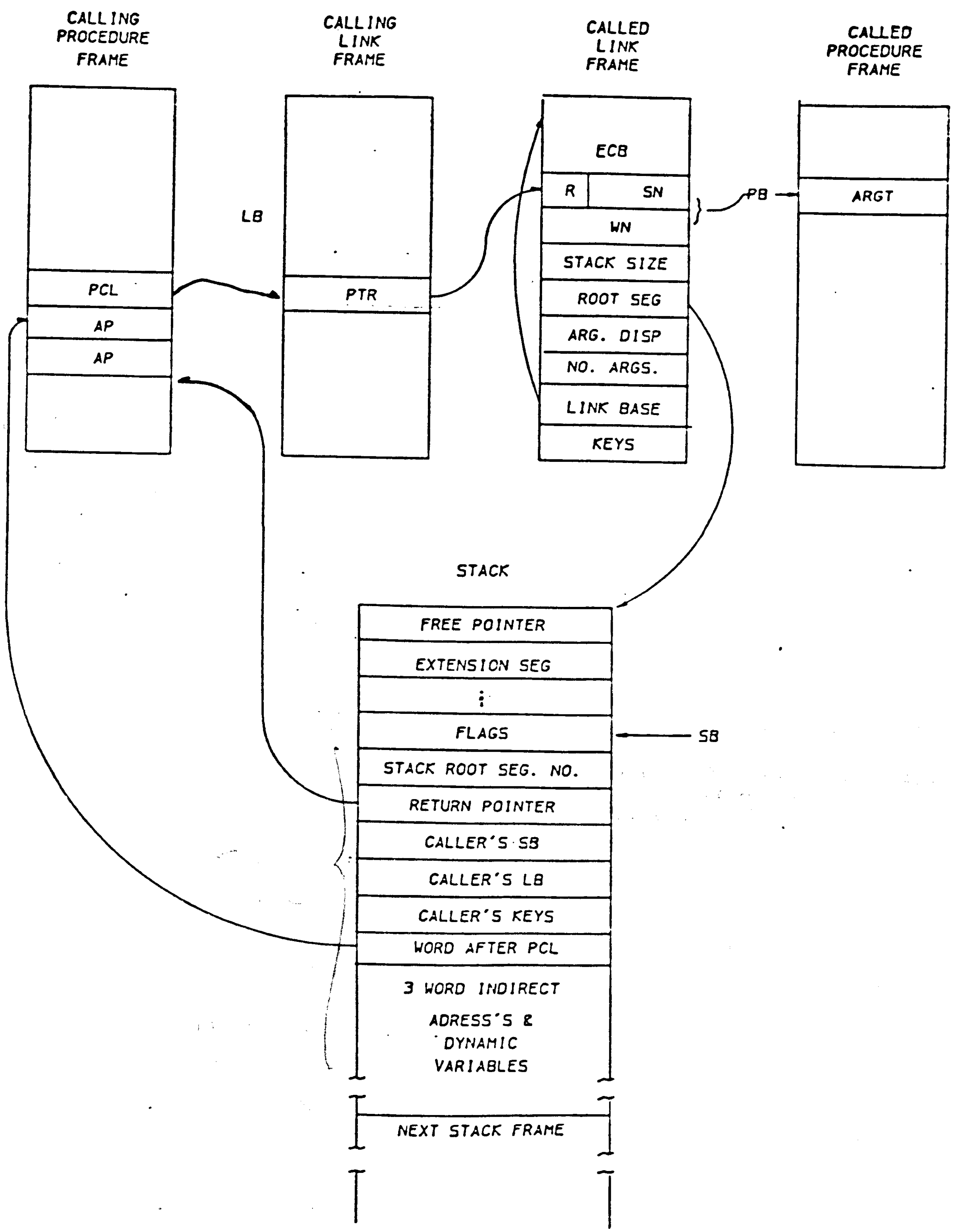
THE ENTRY CONTROL BLOCK

0 1	POINTER TO FIRST EXECUTABLE STATEMENT OF THE CALLED PROGRAM
2	SIZE OF STACK FRAME
3	STACK ROOT SEGMENT NO.
4	ARGUMENT DISPLACEMENT
5	NUMBER OF ARGUMENTS
6 7	LINKAGE BASE ADDRESS OF THE CALLED PROGRAM
10	KEYS FOR THE CALLED PROGRAM
11 17	RESERVED MUST BE ZERO

STACK HEADER AND PCL STACK FRAME FORMAT

0	POINTER TO THE NEXT
1	FREE FRAME
2	POINTER TO THE
3	EXTENSION SEGMENT
0	FLAGS
1	STACK ROOT SEGMENT NUMBER
2	RETURN
3	POINTER
4	CALLER´S STACK
5	BASE
6	CALLER´S LINK
7	BASE
10	CALLER´S KEYS
11	WORD NUMBER AFTER PCL
12	POINTERS TO ARGUMENTS (3 WORD INDIRECT ADDRESSES) AND DYNAMIC VARIABLES

THE PCL MECHANISM



Section 6 - Exception Handling

Objectives: The student will be able to

- o explain what a fault is and how it is handled.
- o describe the actions of ring0 fault handlers.
- o describe the actions of ring3 fault handlers.
- o explain how conditions are handled.
- o track, with VPSD, a dynamic link being snapped.
- o examine DMSTK output to track a particular sequence of events.

FAULT

A FAULT is a condition which has been detected as a result of the currently running software and which requires software intervention. A FAULT may be handled by the current software though most frequently common supervisor code will handle the FAULT (e.g. Page Fault). FAULTs are CPU events which are synchronous with and caused by software.

Two data areas are used:

1). PCB FAULT VECTORs and concealed stack pointers
2). the FAULT TABLEs pointed to by the PCB vectors.

Therefore each process can define its own fault handlers and the concealed stack allows FAULTS to be stacked. The PAGE FAULT has its own vector and only one system-wide handler is used so all PAGE FAULT vectors point to the same place.

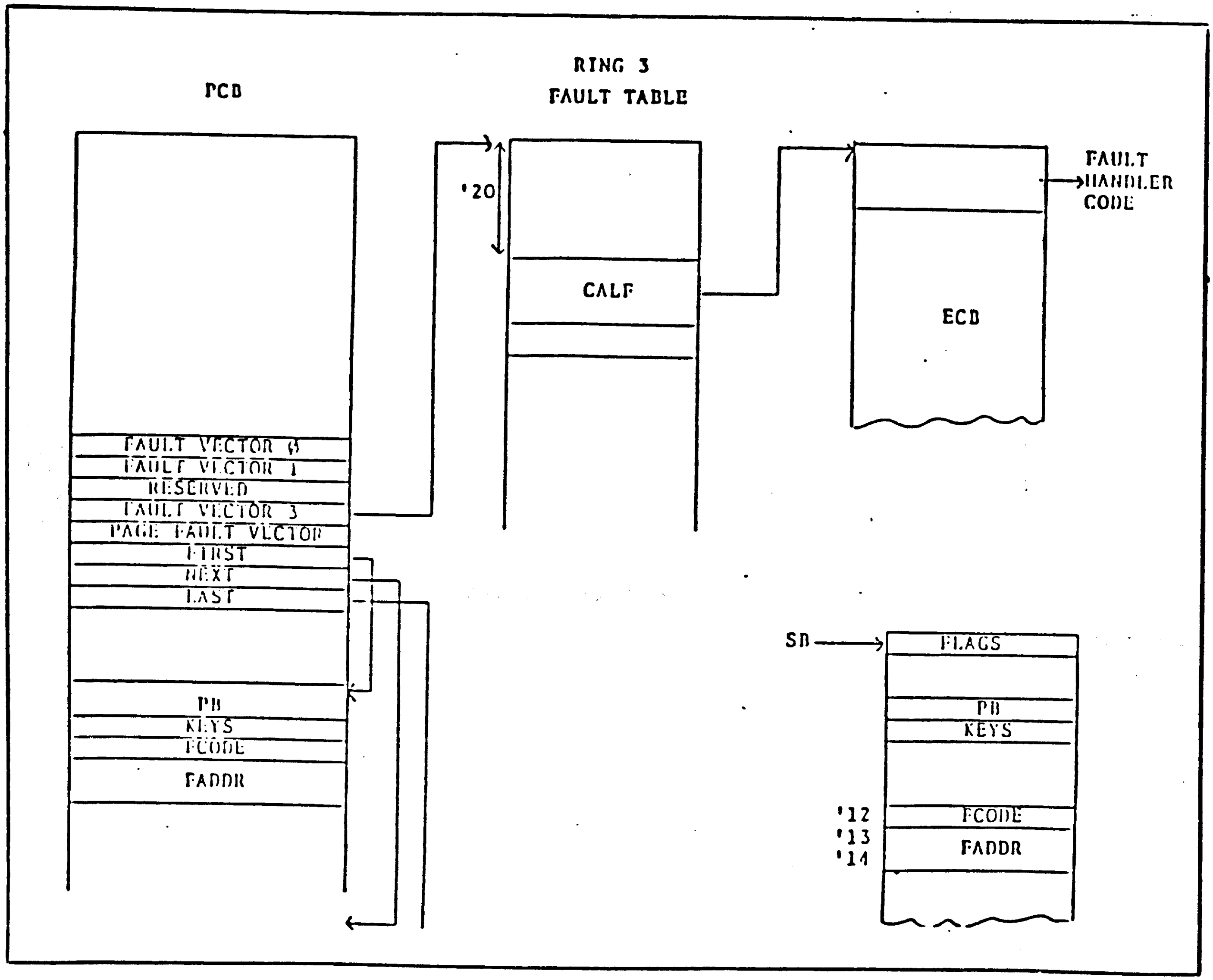
Each FAULT TABLE entry consists of 4 words, of which the first 3 must be a CALF instruction. The CALF (CALl Fault handler) instruction is essentially a PCL (Procedure CaLl) instruction for the various Fault handling routines. The PB and KEYS from the concealed stack are placed in the Fault Handler's stack frame along with other base registers. The Fault Code and Fault Address are placed in words '12,'13, '14 of the Fault Handler's stack. The first word of the new stack frame is set to a value of 1. This is to distinguish the CALF stack frame from the normal PCL stack frame. The ECB (Entry Control Block) addressed by the CALF must not specify any arguments. Return from the fault handler is by normal PRTN instruction.

FAULT PROCESSING

TYPE	OFFSET	RING	SAVED PB	FCODE	FADDR
RESTRICTED INSTRUCTION	0	CURRENT	BACKED	--	--
PROCESS	4	0	CURRENT	ABORT FLAGS	--
PAGE	10	0	BACKED	--	ADDRESS
SVC	14	CURRENT	CURRENT	--	--
UNIMPLEMENTED INSTRUCTION	20	CURRENT	BACKED	CURRENT P COUNTER	EFF ADDRESS
SEMAPHORE OVERFLOW	24	0	BACKED	under = $0 over = $1	SEMAPHORE ADDRESS
ILLEGAL INSTRUCTION	40	CURRENT	BACKED	CURRENT P COUNTER	EFF ADDRESS
ACCESS VIOLATION	44	0	BACKED	--	ADDRESS
ARITHMETIC EXCEPTION	50	CURRENT	CURRENT	EXCEPTION CODE	OPERAND ADDRESS
STACK OVERFLOW	54	0	BACKED	--	LAST STACK SEGMENT
SEGMENT	60	0	BACKED	# too large or Fault Bit	ADDRESS
POINTER	64	CURRENT	BACKED	PTR 1st word	ADDRESS OF PTR

FAULT HANDLING

FAULT OPERATION
(EG. UII in Ring 3)



THE FAULT FRAME HEADER - FFH

When a hardware fault occurs, a stack frame is created by the CALF instruction for the fault handler.

```
dcl 1 ffh based,                        /* standard fault frame header */
       2 flags,
          3 backup_inh bit(1),          /* will be '0'b */
          3 cond_fr bit(1),             /* will be '0' */
          3 cleanup_done bit(1),
          3 efh_present bit(1),         /* will be '0'b */
          3 user_proc bit(1),           /* will be '0'b */
          3 stk_cbits bit(1),           /* will be '0'b */
          3 lib_proc bit(1),            /* will be '0'b */
          3 ecb_cbits bit(1),           /* will be '0'b */
          3 mbz bit(6),
          3 fault_fr bit(2),            /* will be '10'b or '01'b */
       2 root,
          3 mbz bit(4),
          3 seg_no bit(12),
       2 ret_pb ptr,
       2 ret_sb ptr,
       2 ret_lb ptr,
       2 ret_keys bit(16) aligned,
       2 fault_type fixed bin,
       2 fault_code fixed bin,
       2 fault_addr ptr,
       2 hdr_reserved(7) fixed bin,
       2 regs,
          3 save_mask bit(16) aligned,
          3 fac_1(2) fixed bin(31),
          3 fac_0(2) fixed bin(31),
          3 genr(0:7) fixed bin(31),
          3 xb_reg ptr,
       2 saved_cleanup_pb ptr,
       2 pad fixed bin;
```

RING 0 FAULT HANDLERS

The Fault Vector in the user's PCB for RING 0 points to a fault table called FAULT in segment 6. The fault table is defined in PRIMOS>KS>R0FALT.PMA.

The following Fault Handlers exist in Segmment 6:

- PROCESS FAULT
- PAGE FAULT
- UII (UnImplemented Instruction)
- ACCESS VIOLATION
- STACK OVERFLOW
- SEGMENT FAULT
- POINTER FAULT

Any other Fault occurring in RING 0 (e.g. SVC, restricted instruction) will cause the system to HALT.

PROCESS FAULT

1. Check Abort Flags
2. If any Abort Flag is set and aborts are enabled, call PABORT.

SYSTEM ABORT FLAGS - User 1

1 MINALM, ONE MINUTE (MINABT)
Dump any entries in LOGBUF to LOGREC
Update all disk buffers
Decrement auto-logout clocks and logout any USERs out of time.
Process USER 1 message buffer

2 SMLALM, SMLC (SMLCEX) Process SMLC requests

3 NETALM, NETWORK Process network requests (NETUSR at Revision 19)

4 LGIALM, LOGIN (WIRSTK) Lock USER stack, notify user (LOGLCK)

5 WRMALM, WARM START (WRMABT)
Initialize MPC, VERSATEC, and Magnetic Tape
Initialize network and AMLCs.

6 MSGALM, SUPERVISOR MESSAGE (T1OU) Process USER 1 message buffer.

7 CHKALM, Sensor check has occurred.
Turn off como, turn on TTY
Print ´PRIMOS SHUTTING DOWN DUE TO SENSOR CHECK´ at console
Print ´NO COMMANDS ACCEPTED´ at console
Dump any entries in LOGBUF to LOGREC
Flush LOCATE buffers
Logout all users except user 1, NETMAN, and FAM
Shut down disks
Print ´SHUTDOWN COMPLETE´ at console
Halt system

8 Not used

PROCESS FAULT

USER ABORT FLAGS

16 TSEALM, TIME SLICE END (SCHED)
Place process on low priority or eligibility queue

14 TMOALM, FORCED LOGOUT (LOGABT)
Output message 'TIMEOUT', Signal 'LOGOUT$'

13 DISALM, AMLC DISCONNECT LOGOUT OR OPERATOR LOGOUT (LOGABT)
Output message 'FORCE LOGOUT', Signal 'LOGOUT$'

10 IOALM, I/O ALARM Call MTDONE

9 SWIALM, SOFTWARE INTERRUPT (SW$ABT) (formerly QUTALM)

15,12,11 Not Used

SOFTWARE INTERRUPT HANDLING

MOTIVATION

- Due to increased frequency of asynch events at rev 19; more pressure on quit mechanism.
- Ring 0 code had to explicitly inhibit process aborts. Unexpected exit from many ring 0 routines before completion produces non-reliable results.
- Inhibiting quits would disable multiple process abort events.

IMPLEMENTATION

- BREAK$ code reduced to only handle QUIT$.
- SoftWare Interrupt modules for rest of process aborts.
- SWITYP flag word defines which event.
- New mechanism defaults to inhibiting process aborts in ring 0. Enabling quits in ring 0 must now be explicitly performed.

SOFTWARE INTERRUPT HANDLING - Routines and Variables

BREAK$ - enable/disable QUIT$ aborts in ring 0

SW$INT - process abort interrupt enable/disable control

SETSWI - store event bit in PUDCOM.SWITYP

SETABT - set user's abort flags

SW$ABT - fault handler for process aborts

SWFIM_ - handles deferred ring 0 aborts on return to outer ring

SW$RST - called by SWFIM_ to reset ROSWIN, ROQUIT

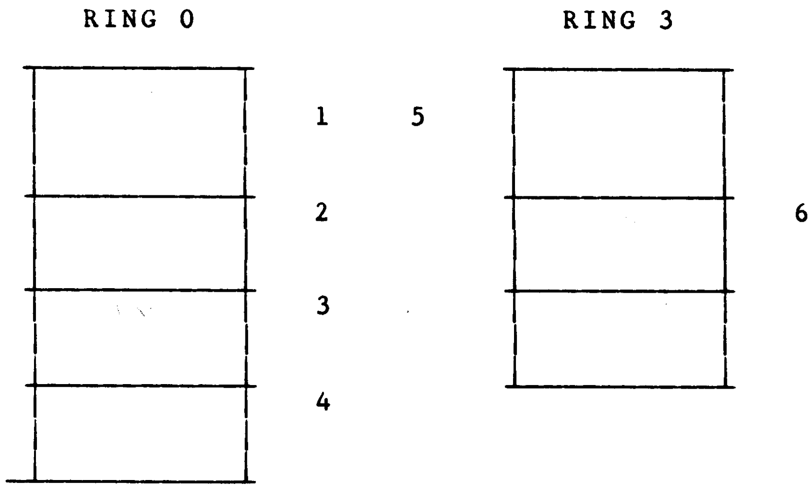
Variables: SWITYP -

```
QUTINT  EQU   '100000      QUIT
CPUINT  EQU   '40000       CPU TIME WATCHDOG
TIMINT  EQU   '20000       REAL TIME WATCHDOG
LOGINT  EQU   '10000       FORCED LOGOUT
LONINT  EQU   '4000        LOGOUT NOTIFICATION
CPSINT  EQU   '2000        CROSS PROCESS SIGNALLING
IPCMWI  EQU   '1000        IPC MESSAGE WAITING
WRMINT  EQU   '400         WARMSTART SOFTWARE INTERRUPT
```

ROSWIN - ring 0 software interrupt enable word
ROQUIT - ring 0 quit enable counter

PROCESS ABORT HANDLING

RING 0, INTERRUPT DISABLED



1)

2)

3)

4)

5)

6)

OTHER RING 0 FAULTS

UII FAULT in ring 0 will HALT the machine except when operating on a P400/350 using XVRY, ZMV, ZMVD, ZFIL, and ZCM which are simulated in a routine called ROUII in segment 6.

SEMAPHORE FAULT Save semaphore status information and HALT the system.

ACCESS VIOLATION call SIGNAL$ called to output the message "ACCESS VIOLATION RAISED AT ...."

STACK OVERFLOW call STKOVF, SIGNAL$ 'STACK_OVF$', message "STACK-OVF$ RAISED AT ...."

SEGMENT FAULT call GETSEG to either allocate a segment or call SIGNAL$ to output the message "ILLEGAL SEGNO$ RAISED AT ...."

POINTER FAULT - Ring 0

1). Save user state
2). Pick up faulting pointer
3). Return if pointer is greater or equal 0
4). Erase fault bit
5). Error message if pointer is equal 0, or invalid
6). Call SNAP$3 to get new pointer
7). Snap link
8). If not found error message

POINTER FAULT outputs the message "POINTER-FAULT$ RAISED AT ...."

RING 3 FAULTS

The fault vector in the user's PCB for ring 3 points to a fault table called R3FALT in segment 13.

The following fault handlers exist in segment 13:

- RESTRICTED INSTRUCTION FAULT
- SVC FAULT
- UII FAULT
- ILLEGAL INSTRUCTION FAULT
- ARITHMETIC FAULT
- STACK OVERFLOW FAULT
- POINTER FAULT

Any other fault occuring in ring 3 is handled by the ring 0 fault handlers.

RESTRICTED INSTRUCTION FAULT

Call PTRAP in ring 0

1). Read violating instruction and analyze.
2). If illegal or HALT instruction call SIGNAL$ to output the message 'PROGRAM HALT AT .....'
3). Simulate trapped I/O instructions for
    - System console, CRTs
    - Paper tape reader/punch
    - Card reader
    - Control panel

SVC

Enter SVC fault handler to initiate SVC and pass arguments.

UII FAULT

Enter UII routine in segment 13 to software emulate the instruction.

ILLEGAL INSTRUCTION FAULT

Enter illegal instruction fault handler which signals 'ILLEGAL-INST$'.

ARITHMETIC FAULT

Enter arithmetic fault handler which signals ARITH$ condition.

RING 3 FAULTS

STACK OVERFLOW FAULT

Call STKOVF. (Automatic Ring 3 Stack Extension)

Examine stack frame prior to fault frame and determine stack root segment.
If root is ´6002 then STK_EX is called.
Otherwise condition ´STACK_OVF$´ is signalled as before.

STK_EX
Attempts to get a DTAR 2 dynamic segment.
If not possible calls FATAL$.
Otherwise fixes up stack extension ptr to point to new segment, and returns.

POINTER FAULT

1). Save user state

2). Clear fault bit

3). If bad pointer - signal POINTER-FAULT$

4). Call LN_SLIB to initiate the search to snap the link:

a) Call SNAP$0 to check if the routine is a R0 gate routine, and if it is, return ECB address.
b) Call SNAP$3 to check if the routine is an ´All Rings Callable´ routine, and if it is, return ECB address.
c) Call LN_EPF or LN_STAT, based on user´s search list, to check if the routine is in an EPF library or a static mode library, and if it is, return ECB address.

5). If ECB address found, replace faulty pointer (i.e., snap the link) and execute the PCL again.

If ECB address not found, signal LINKAGE_FAULT$.

If an error occurred while attempting to resolve the faulty reference, signal LINKAGE_ERROR$.

DIRECT ENTRANCE CALLS

The direct entrance call (DEC) mechanism provides a form of dynamic linking using the standard Procedure CaLl (PCL) instruction (V-mode only) and the indirect memory address pointer. The purpose of the DEC is to provide an efficient mechanism for application and system programs to call procedures that are part of the operating system or shared libraries. The DEC provides a mechanism to share a single copy of a procedure among all users on the system. These procedures do not have to be relinked for a different revision of PRIMOS, since the address linkage to the procedure is not made until execution time.

A special form of object module, called a DYNT, is created by assembling a PMA program that has the form:

```
SEG
SYML
DYNT  procedure_name
END
```

When the SEG or BIND loaders encounter this structure they put an indirect pointer in the link frame of the calling procedure that has the fault bit set which points to a location in the procedure area where SEG or BIND has put the name of the direct entrance call and the length of the name.

At execution time when the call is made, the fault bit causes the hardware to detect a pointer fault and enter the pointer fault handler. The pointer fault handler attempts to resolve the address linkage to the called procedure by searching lists of ECBs (entry points) to the direct entrance callable routines. If the ECB is found, the address pointer to the procedure is stored back in the pointer that originally caused the fault, the fault bit is erased and the call is reexecuted (without the fault).

VITAL STATS FOR DIRECT ENTRANCE CALLS IN RING 3

Ring 0	
Hash Table Generator	- PRIMOS>HASH>GENERATE_HASH_TABLE.SPL
Hash Table Entry Names	- PRIMOS>KS>GATE_TABLE_HASH
Hash Table	- PRIMOS>KS>GATE_HTB.PMA
Routines	- PRIMOS>R3S>R3FALT.PMA (pointer fault handler) - PRIMOS>R3S>LN_SLIB.PLP - PRIMOS>KS>SNAP$0.PLP - PRIMOS>R3S>SEARCH_HASH_TABLE$.PLP (SRCH$HTB) - PRIMOS>R3S>FIND$BKT.PLP - PRIMOS>R3S>HASH_UID.PLP
Memory Location	- Segment 5
All Rings Callable	
Hash Table Generator	- PRIMOS>HASH>GENERATE_HASH_TABLE.SPL
Hash Table Entry Names	- PRIMOS>R3S>RING3_ENTRY_TABLE_HASH
Hash Table	- PRIMOS>R3S>R3ENTS.PMA
Routines	- PRIMOS>R3S>R3FALT.PMA (pointer fault handler) - PRIMOS>R3S>LN_SLIB.PLP - PRIMOS>R3S>SNAP$3.PMA - PRIMOS>R3S>SEARCH_HASH_TABLE$.PLP (SRCH$HTB) - PRIMOS>R3S>FIND$BKT.PLP - PRIMOS>R3S>HASH_UID.PLP
Memory Location	- Segment 13
Static Mode Libraries	
Hash Table Generator	- DIRECV>HASHER.FTN
Hash Table Entry Names	- HTAB (Each library that is to be shared has a table called HTAB in its source file UFD.)
Hash Table	- HTAB (DIRECV>R3POFH.PMA -- There will be a copy of this procedure, each with its own HTAB for each shared library installed.)
Routines	- PRIMOS>R3S>R3FALT.PMA (pointer fault handler) - PRIMOS>R3S>LN_SLIB.PLP - PRIMOS>R3S>LN_STAT.PLP (LIBTBL) DIRECV>R3POFH.PMA (HTAB)
Memory Location	- Segment 2xxx

VITAL STATS FOR DIRECT ENTRANCE CALLS (CONT'D)

EPF Libraries

Hash Table Generator	- BIND loader
Hash Table Entry Names	- Input to the BIND loader
Hash Table	- Internal to Library
Routines	- PRIMOS>R3S>R3FALT.PMA (pointer fault handler)
	- PRIMOS>R3S>LN_SLIB.PLP
	- PRIMOS>R3S>LN_EPF.PLP
	- PRIMOS>R3S>EPF_SRCH.PLP
	- PRIMOS>R3S>KTRAN$.PMA
Memory Location	- Segment 4xxx

PRIMOS HASH TABLE FORMAT

```
                           ---------------------------------
HASH TABLE HEADER ->|  VERSION OF HASHING FUNCTION  |
                           ---------------------------------
                           |     SIZE OF PRIME AREA          |
                           ---------------------------------
                           |   NUMBER OF ACTIVE ENTRIES      |
                           ---------------------------------
                           |    TOTAL NUMBER OF ENTRIES      |
                           ---------------------------------
                           |   POINTER TO FIRST NAME IN      |
                           |          NAME TABLE             |
ENTRY FORMAT ------>---------------------------------
                           |          POINTER TO             |
                           |             NAME                |
                           ---------------------------------
                           |    LINK TO OVERFLOW ENTRY       |
                           ---------------------------------
                           |           RESERVED              |
                           ---------------------------------
                           |          POINTER TO             |
                           |    DATA (e.g., ECB ADDRESS)     |
                           ---------------------------------
```

STATIC MODE LIBRARIES - LIBTBL

LIBTBL is a table that contains address pointers to the search routines for the various static mode libraries. Entries in LIBTBL are generally made according to the package number.

```
LIBTBL -->  --------------------------------
           |    A(ECB) FOR THE R3POFH       |
           |       FOR PACKAGE #1           |
            --------------------------------
           |    A(ECB) FOR THE R3POFH       |
           |       FOR PACKAGE #2           |
            --------------------------------
            ~~                            ~~


            ~~                            ~~
            --------------------------------
           |    A(ECB FOR THE R3POFH        |
           |      FOR PACKAGE #32           |
            --------------------------------
           |           7777                 |
           |             0                  |
            --------------------------------
```

CONDITION MECHANISM

MOTIVATION

- system software error handling
- manage reentrant/recursive command environment
- user program error (and event) handling
- support ANSI PL/1 condition mechanism

IMPLEMENTATION

- extended stack header
- on-unit descriptor block (on stack)
- condition frame header (on stack)
- fault frame header (on stack)

CONDITION MECHANISM - DEFINITIONS

CONDITION - an unscheduled event

ON-UNIT - a procedure to handle an event

MAKE ON-UNIT - turn on event handler for this activation

REVERT ON-UNIT - turn off event handler for this activation

SIGNAL - telling the world the event happened

RAISE - procedure which searches the stack for the ON-UNIT

CRAWL_ - procedure which switches from inner ring to ring 3 stack

NON-LOCAL-GOTO - a goto to a predefined label not in this activation

DEFAULT ON-UNIT - one example of system use of condition mech.

THE EXTENDED STACK FRAME HEADER - EFH

Any procedure which is to create one or more on-units must reserve space in its stack frame for an extension that contains descriptive information about those on-units. Most of the compilers that support the condition mechanism will automatically allocate this extra space.

```
dcl 1 sfh based,                      /* stack frame header */
      2 flags,
         3 backup_inh bit(1),         /* inhibit crawlout-backup of pb */
         3 cond_fr bit(1),
         3 cleanup_done bit(1),
         3 efh_present bit(1),        /* extension to frame is here */
         3 user_proc bit(1),
         3 stk_cbits bit(1),          /* stack has valid cond bits */
         3 lib_proc bit(1),           /* is a library procedure */
         3 ecb_cbits bit(1),          /* ecb has valid cond bits */
         3 mbz bit(6),
         3 fault_fr bit(2),           /* '00'b -> pcl frame */
      2 root,
         3 mbz bit(4),
         3 seg_no bit(12),            /* seg number of root of stack */
      2 ret_pb ptr,                   /* caller's return point */
      2 ret_sb ptr,                   /* caller's stack frame */
      2 ret_lb ptr,                   /* caller's link frame */
      2 ret_keys bit(16) aligned,     /* caller's keys */
      2 after_pcl fixed bin,          /* relp to <pcl_instr> + 2 */
      2 hdr_reserved(8) fixed bin,
      2 owner_ptr ptr,                /* ptr to ecb that created frame */
      2 tempsc(8) fixed bin,          /* standard shortcall temps */
      2 onunit_ptr ptr,               /* first ODB on the chain */
      2 cleanup_onunit_ptr ptr,       /* null if no cleanup onunit */
      2 next_efh ptr,                 /* points to next exten headers */
      2 spl_lib_scratch(6) fixed bin,
      2 cond_bits bit(16) aligned; /* PL1 condition enable bits */
```

THE ON-UNIT DESCRIPTOR BLOCK - ODB

Each on-unit created by an activation is described to the condition mechanism by a descriptor block (except for CLEANUP$). These descriptor blocks for a given activation are chained together in a simple linked list.

```
dcl 1 onub based,                  /* standard onunit block */
      2 ecb_ptr ptr,               /* ecb to call on invocation */
      2 next_ptr ptr,              /* next ODB in this activation */
      2 flags,
        3 not_reverted bit(1),     /* ignore if '0'b */
        3 is_proc bit(1),          /* '0'b->is begin block(pl1 onunit) */
        3 specify bit(1),          /* check onub.specifier if on */
        3 snap bit(1),             /* snap option requested */
        3 mbz bit(12),
     2 pad fixed bin,              /* must be 0 */
     2 cond_name_ptr ptr,          /* ptr to char(32) var cond name */
     2 specifier ptr;              /* e.g. file desc ptr for "endfile" */
```

THE CONDITION FRAME HEADER - CFH

SIGNL$ takes its own standard PCL stack frame and turns it into a condition frame for the condition being signalled.

```
dcl 1 cfh based,                    /* standard condition frame header */
      2 flags,
         3 backup_inh bit(1),     /* will be '0'b */
         3 cond_fr bit(1),        /* will be '1'b */
         3 cleanup_done bit(1),
         3 efh_present bit(1),    /* will be '0'b */
         3 user_proc bit(1),      /* will be '0'b */
         3 stk_cbits bit(1),      /* will be '0'b */
         3 lib_proc bit(1),       /* will be '0'b */
         3 ecb_cbits bit(1),      /* will be '0'b */
         3 mbz bit(6),
         3 fault_fr bit(2),       /* will be '00'b */
      2 root,
         3 mbz bit(4),
         3 seg_no bit(12),
      2 ret_pb ptr,
      2 ret_sb ptr,
      2 ret_lb ptr,
      2 ret_keys bit(16) aligned,
      2 after_pcl fixed bin,
      2 hdr_reserved(8) fixed bin,
      2 owner_ptr ptr,
      2 cflags,
         3 crawlout bit(1),
         3 continue_sw bit(1),
         3 return_ok bit(1),
         3 inaction_ok bit(1),
         3 specifier bit(1),
         3 ring_limit bit (2),       /*0 = no ring limit
                                        1 = ring 1 limit
                                        2 = ring 0 limit
                                        3 = ring 3 limit for signals*/
         3 sou_crash bit (1),        /*sou crash indicator*/
         3 sou_comp_hndld bit (1), /*sou hndld not to df_unit*/
         3 mbz bit(7),
      2 version fixed bin,           /* init(1) */
      2 cond_name_ptr ptr,
      2 ms_ptr ptr,              /* machine state at time of signal */
      2 info_ptr ptr,
      2 ms_len fixed bin,
      2 info_len fixed bin,
      2 saved_cleanup_pb ptr;
```

DMSTK OUTPUT

```
OK, seg sleep
This is SLEEP.FTN, going to sleep for one minute      /* normal
This is SLEEP.FTN, finished sleeping, exiting         /* execution


OK, seg sleep
This is SLEEP.FTN, going to sleep for one minute      /* control P
                                                      /* typed
QUIT.


OK, DMSTK -ALL -ON_UNITS
Backward trace of stack from frame 1 at 6002(3)/7756.

STACK SEGMENT IS 6002.

(1) 007756: Owner=  (LB= 13(0)/13540).                /* STD$CP
    Called from 13(3)/110567; returns to 13(3)/110573./* (INTERNAL_
                                                      /* EXECUTER)

(2) 006700: Owner=  (LB= 13(0)/112404).               /* CP_ITER
    Called from 13(3)/107765; returns to 13(3)/107771./* (LIGASE)

(3) 004440: Owner=  (LB= 13(0)/112404).               /* CP_ITER
    Called from 13(3)/10516; returns to 13(3)/10536.

(4) 003706: Owner=  (LB= 13(0)/13540).                /* STD$CP
    Called from 13(3)/3123; returns to 13(3)/3135.
    Onunit for "CLEANUP$" is 13(3)/14541.
    Onunit for "STOP$" is 13(3)/14341.
    Onunit for "SUBSYS_ERR$" is 13(3)/14361.

(5) 003370: Owner=  (LB= 13(0)/4162).                 /* LISTEN_
    Called from 13(3)/104526; returns to 13(3)/104532.
    Onunit for "CLEANUP$" is 13(3)/4714.
    Onunit for "ANY$" is 13(3)/77424.
    Onunit for "LISTENER_ORDER$" is 13(3)/4754.
    Onunit for "SETRC$" is 13(3)/4734.
    Onunit for "REENTER$" is 13(3)/4774.

(6) 003344: Owner=  (LB= 13(0)/104142).               /* COMLV$
    Called from 13(3)/63426; returns to 13(3)/63430.

(7) 002560: Owner=  (LB= 13(0)/66176).                /* DF_UNIT_
    Called from 13(3)/52601; returns to 13(3)/52605.

(8) 002460: Owner=  (LB= 13(0)/52316).                /* RAISE_
    Called from 13(3)/51651; returns to 13(3)/51663.
```

DMSTK OUTPUT (CONT'D)

```
(9) 002332: CONDITION FRAME for "QUIT$"; returns to 13(3)/56625.
    Condition raised at 6(0)/3421; LB= 6(0)/3300, Keys= 014000
    (Crawlout to 4001(3)/1043; LB= 4002(0)/177400.)
    Inner ring fault: type "PROCESS" (4); code= 000200; addr= 0(0)/0
    Registers at time of fault in inner ring:
                    Save Mask= 000000;  XB= 6(0)/1402
       GR0      0     0          0   GR1     0     0          0
     L,GR2      0     0          0 E,GR3     0     0          0
       GR4      0     0          0 Y,GR5     0     0          0
       GR6      0     0          0 X,GR7     0     0          0
     FAR0 0(0)/0            FLR0           0 FR0   0.00000000E 00
     FAR1 0(0)/0            FLR1           0 FR1   0.00000000E 00

(10) 002130: Owner=  (LB= 13(0)/56236).                  /* CRFIM_
    Called from 4001(3)/1043; returns to 4001(3)/1043.

STACK SEGMENT IS 4001.          /** CONTROL P TYPED HERE **/

(11) 001174: Owner=  (LB= 4002(0)/177400).               /* SLEEP.FTN
    Called from 4000(3)/61677; returns to 4000(3)/61701.

STACK SEGMENT IS 4000.

(12) 150062: Owner=  (LB= 4000(0)/61364).                /* SEG(VRUNIT)
    Called from 4000(3)/1723; returns to 4000(3)/1725.
    Proceed to this activation is prohibited.

(13) 150012: Owner=  (LB= 4000(0)/5064).                 /* SEG(MAIN)
    Called from 4000(3)/1100; returns to 4000(3)/1102.
    Onunit for "CLEANUP$" is 4000(3)/62470.

(14) 150000: Owner=  (LB= 4002(0)/177400).               /* INVALID FRAME
    Called from 0(0)/177776; returns to 0(0)/0.          /* SETUP BY SEG
```

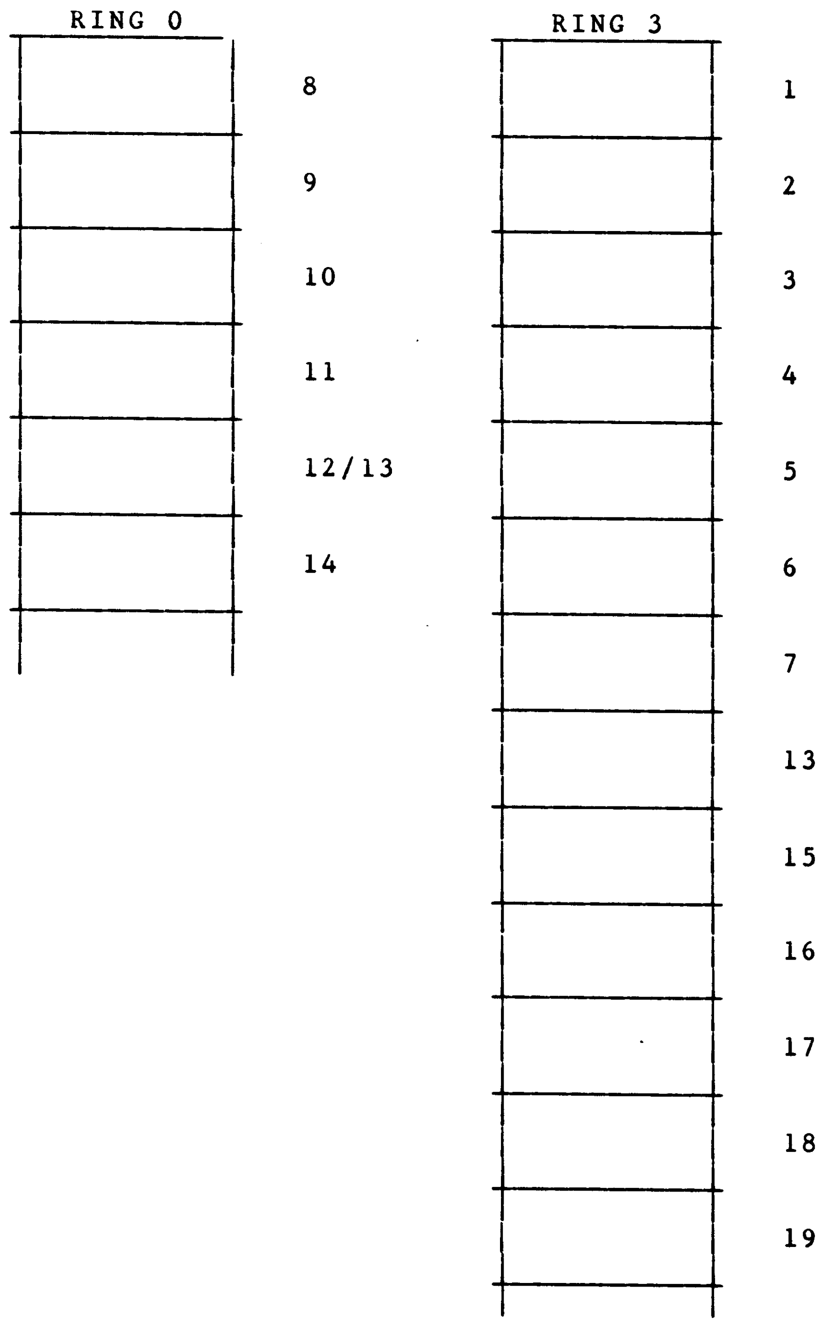
DMTSK OUTPUT (CONT'D)

STACK SEGMENT IS 6002.

```
(15) 001666: Owner=  (LB= 13(3)/31746).                   /* INVKSM_
    Called from 13(3)/13174; returns to 13(3)/13216.
    Onunit for "CLEANUP$" is 13(3)/32433.
    Onunit for "ANY$" is 13(3)/32413.

(16) 001506: Owner=  (LB= 13(0)/13540).                   /* STD$CP
    Called from 13(3)/12114; returns to 13(3)/12120.      /* (SM_EXECUTER)

(17) 000764: Owner=  (LB= 13(0)/13540).                   /* STD$CP
    Called from 13(3)/3123; returns to 13(3)/3135.
    Onunit for "CLEANUP$" is 13(3)/14541.
    Onunit for "STOP$" is 13(3)/14341.
    Onunit for "SUBSYS_ERR$" is 13(3)/14361.

(18) 000446: Owner=  (LB= 13(0)/4162).                    /* LISTEN_
    Called from 13(3)/152454; returns to 13(3)/152460.
    Onunit for "CLEANUP$" is 13(3)/4714.
    Onunit for "ANY$" is 13(3)/77424.
    Onunit for "LISTENER_ORDER$" is 13(3)/4754.
    Onunit for "SETRC$" is 13(3)/4734.
    Onunit for "REENTER$" is 13(3)/4774.

(19) 000440: Owner=  (LB= 13(0)/152074).                  /* INFIM_
    Called from 1(0)/152456; returns to 1(0)/0.
```

SOFTWARE INTERRUPT HANDLING and CONDITIONS

RING 0, INTERRUPTS ENABLED



SOFTWARE INTERRUPT HANDLING and CONDITIONS

RING 0, INTERRUPTS ENABLED

1)

8)

2)

9)

3)

10)

4)

11)

5)

12)

6)

13)

7)

14)

SOFTWARE INTERRUPT HANDLING and CONDITIONS (CONT'D)

RING 0, INTERRUPTS ENABLED

15)

16)

17)

18)

19)

LOGOUT$ CONDITION

A forced logout will result in SETABT setting the DISALM PCB abort flag or the TMOALM PCB abort flag. This will be intercepted by PABORT, which in turn calls LOGABT.

There are five cases:

(1) forced logout (either by operator or AMLC disconnect)
(2) cpu time limit exceeded
(3) inactivity time limit exceeded
(4) login time limit exceeded
(5) grace period to process LOGOUT$ condition exceeded

When (1) - (4), LOGABT will

(a) inhibit process aborts
(b) set login time limit to (grace period)
(c) call SETSWI(LOGINT)
(d) call SETABT(SWIALM)
(e) enable process aborts
(f) call SW$ABT to signal LOGOUT$

When (5), log the process out immediately.

LOGOUT$ CONDITION - GRACE PERIOD

If the user process has a handler for LOGOUT$, then there will be (grace_period) minutes left in which to tidy up the environment before the final logout.

Otherwise, DF_UNIT_ will simply print the error message and call LOGOU$.

```
when (login_limt)
     call ioa$('login time limit exceeded')
when (cpu_limit)
     call ioa$('cpu time limit exceeded')
when (timeout)
     call ioa$('maximum inactive time limit exceeded')
otherwise
     call ioa$('forced logout')
call logou$;
```

LOGOU$ (LOGOUT)

call internal routine LOGMSG to print message to system console and user terminal.

If a phantom, queue Logout Notification (LON) message to spawner.

CRAWLOUT

Crawlout occurs when the end of an inner ring stack has been reached by the condition mechanism without handling the condition.

Control always orginates in an outer ring, the end of an inner ring stack is threaded to an outer ring stack. The condition mechanism continues the stack search across the connection and back down the outer ring stack. Crawlout is the mechanism which copies the information describing the condition to the outer ring and resignals.

When RAISE reaches the end of the inner ring stack, it returns to SIGNL$ with the CRAWLOUT_NEEDED flag set, a pointer to the last stack frame on the inner ring (CRAWL_FRAME) and a pointer to the most recent inner ring stack frame in which the registers are saved.

SIGNL$ calls CRAWL_ defining the crawlout fault interceptor module (CRFIM_). The stack frame on the outer ring is the target frame.

CRAWL_ checks the space needed in the outer ring stack for the target ring stack and copies the neccessary information into the target stack. The return information in CRAWL_FRAME is adjusted to appear as though it was called from the target frame.

UNWIND is called to unwind the stacks. A procedure return is then invoked to CRFIM_.

CRFIM_ calls SIGNL$ to signal the condition in the outer ring and the on-unit will invoke the next LISTEN_ level.

Section 7 - Command Environment

Objectives: The student will be able to

- o describe how a command is executed

EXTENDED FEATURES

- Command processor enhanced to support following extended features:

    simple iteration
    wildcard expansion
    treewalking
    name generation
    special reserved arguments

- All above are processed by c.p. itself.

- Enabling of individual features may be selected in various ways:

  CPL - defined to have c.p. do simple iteration only

  Static Programs - all features enabled unless special names:
  NW$ - no wildcard or equalname
  NX$ - only simple iteration

  EPF - enabled features specified at BIND time and stored in file

  Internal Commands - enabled features specified in internal command table

EXTENDED FEATURES

CP_ITER	- main routine which processes extended features - makes three passes over command line to verify syntax, expand iteration, process options
Pass I	- parses command line into 2 level tree - each node represents a token - 2nd level for simple iteration tokens
Pass II	- repeated while iteration in progress - convert tree into simple threaded list - expand dot products - call DCOD_ITR to find type of token (e.g. wildcard, wildtree, control, equalname)
Pass III	- repeated while iteration in progress - verify only one wildcard/tree per line - find location of wild tokens - if wildtree call ITR_WLDT - if wildcard call ITR_WLDC - if no wilds call LIGASE - free all temporary storage
ITR_WLDT	- expands wild trees - uses control args if supplied - calls ITR_WLDC if wildcards, or 'executer' to execute each match - recurses when required
ITR_WLDC	- expands wild cards - uses control args if supplied - asks user for verification if reqd - calls 'executer' to execute each match
EQUAL$P	- special routine for c.p. - splits pathnames into dir and entry - calls EQUAL$ to match names
EQUAL$	- parse generation pattern components - process 'commands' in components - build generated name by concatenation

EXTENDED FEATURES

LIGASE (internal to CP_ITER)
- follows assembled node list concatenating tokens to form command line
- calls EQUAL$P to process name generation
- call 'executer' routine to execute line

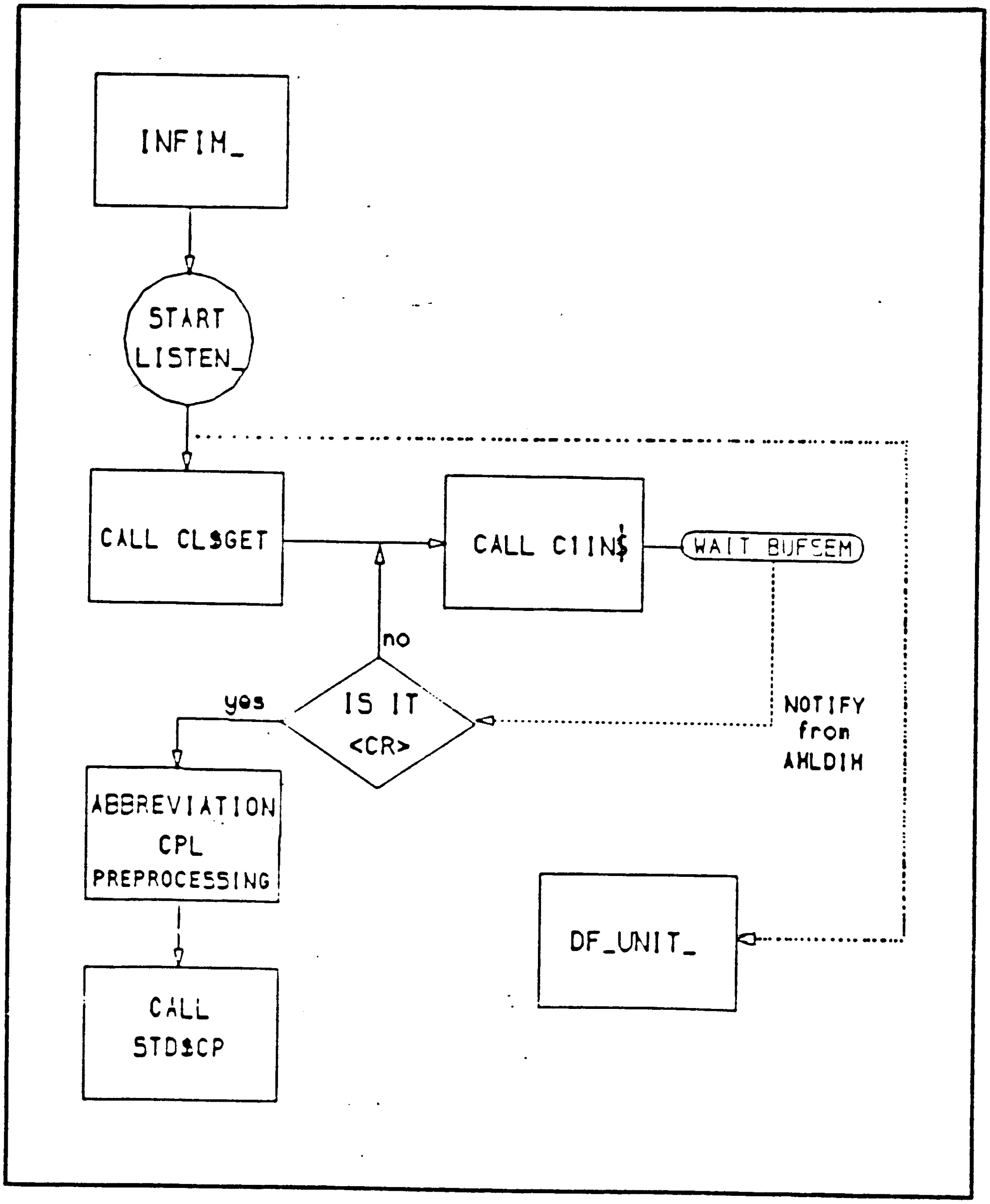
SM_EXECUTER (internal to STD$CP)
- executes static mode command
- calls INVKSM_

CPL_EXECUTER (internal to STD$CP)
- executes CPL command
- calls ICPL_

INTERNAL_EXECUTER (internal to STD$CP)
- executes an internal command
- calls appropriate routine directly

RUN_EXECUTER (internal to STD$CP)
- executes an EPF
- calls EPF$MAP to map in procedure
  EPF$ALLC to allocate linkage
  EPF$INIT to initialize linkage
  EPF$INVK to execute EPF

BUILDING THE COMMAND LINE



COMMAND LINE DATA - CLDATA

This static structure defines the current state of a process' ring3 command environment. The location of this static data block is defined in both the ring0 and ring3 operating system loads.

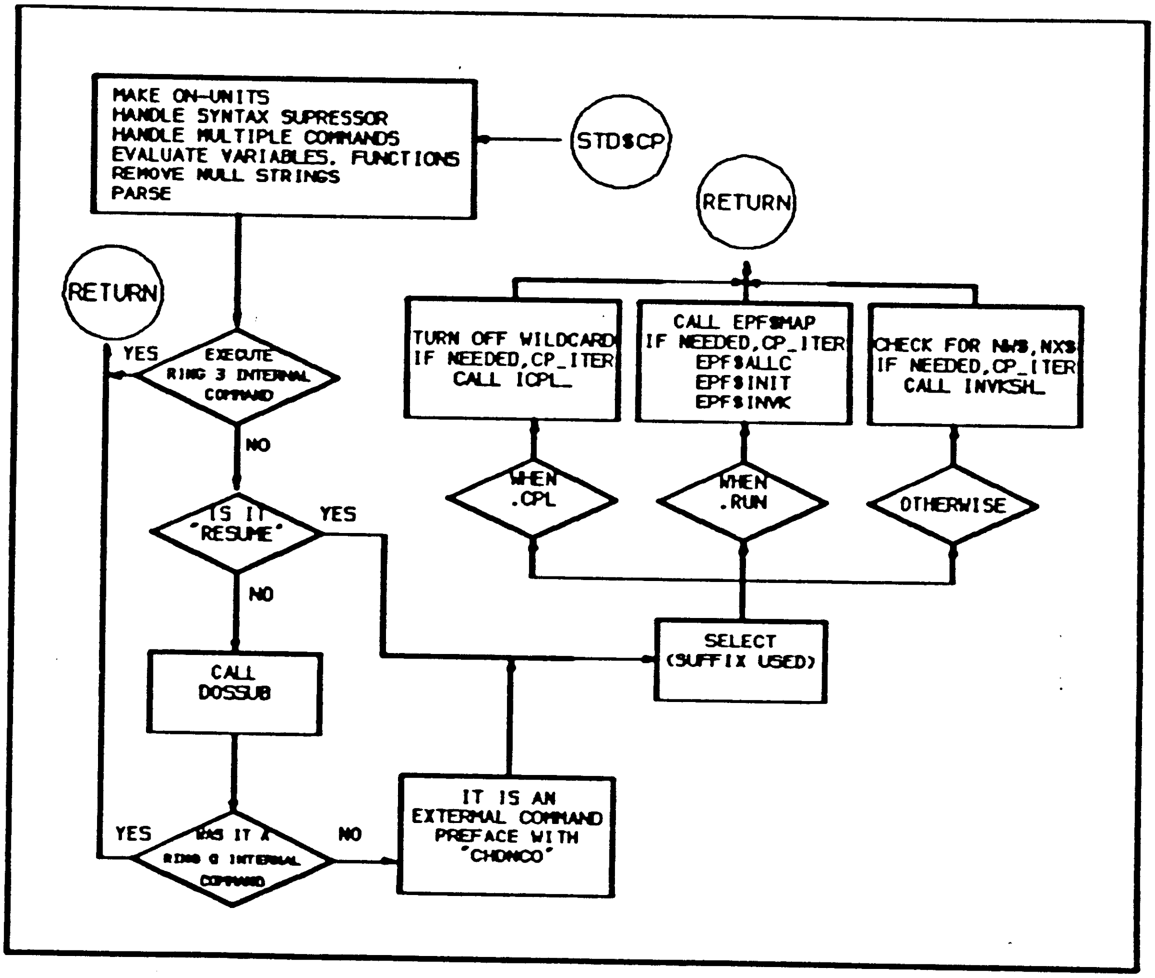
6002/12

```
dcl 1 cldata ext static,                     /* command loop data */
      2 exit_sb ptr options(short),           /* to find stack_3 at exit
                                                from SM procs, PUSHED */
      2 exit_lb ptr options(short),           /* to find stack_3 at exit
                                                from SM procs, PUSHED */
      2 user_number fixed bin(15),            /* system user id */
      2 svcsw bit(16) aligned,                /* virtual svc control */
      2 flags,
         3 ready_on bit(1),                   /* enable ready msgs */
         3 ready_br bit(1),                   /* short ready msgs */
         3 dbg_mode bit(1),                   /* '1'b->debugger in use */
         3 abbrev_on bit(1),                  /* '1'b->use abbrev cmd proc */
         3 sm_used bit(1),                    /* '1'b->SM used at this lvl */
         3 abbrev_ver bit(1),                 /* '1'b->print expand cmd ln */
         3 mbz bit(10),
      2 com_line char(160) var,                  /* command line buffer */
      2 com_line_size fixed bin(15),             /* (size(com_line)-1)*2) */
      2 com_parse_data fixed bin(15),          /* parse data for SM rdtk$$ */
      2 prog_session_depth fixed bin(15),/* breadth of command env. */
      2 sm_fault_fr ptr options(short),        /* to SM ffh at this lvl */
      2 prev_smff ptr options(short),          /* to SM ffh of prev lvl */
      2 level fixed bin(15),                   /* current cmd lvl,PUSHED */
      2 rvec,                                  /* the sm state vector */
         3 start_addr fixed bin(15),
         3 end_addr fixed bin(15),
         3 keys bit(16) aligned,
         3 pb ptr options(short),
         3 sb ptr options(short),
         3 lb ptr options(short),
         3 regs,                              /* in rsav format */
            4 save_mask bit(16) aligned,
            4 fac1(2) fixed bin(31),
            4 fac0(2) fixed bin(31),
            4 genr(0:7) fixed bin(31),
            4 xb ptr options(short),
```

COMMAND LINE DATA - CLDATA (CONT'D)

```
      2 abbrev,                          /* data for the abbrev c.p. */
         3 segptr ptr options(short), /* ptr to live abbrev tbl seg */
         3 treename char(80) var,        /* abbrev file */

      2 sm_err_code fixed bin(15),       /* for static mode */
      2 cpu_secs fixed bin(15),          /* cpu meter, seconds */
      2 cpu_ticks fixed bin(15),         /* cpu meter, secs/330 */
      2 io_secs fixed bin(15),           /* io meter, seconds */
      2 io_ticks fixed bin(15),          /* io meter, secs/330 */

/* Command processor to call upon. Must agree with DCL for
   entry variable STD$CP in the routine INIT$3. */

      2 command_processor entry (char(*) var, fixed bin(15),
        fixed bin(15), 1, 2 bit(1) aligned, 2 bit(1), 2 bit(14),
        ptr options(short), ptr options(short)) variable,

/* Command line reader to call upon. Must agree with DCL for
   entry variable CL$GET in the routine INIT$3. */

      2 command_line_reader entry (char(*) var, fixed bin,
        fixed bin) returns (bit(16) aligned) variable,

/* Command prompt routine to call upon. Must agree with DCL for
   entry variable READY$ in the routine INIT$3. */

      2 command_prompt entry (bit(16) aligned, fixed bin) variable,

      2 ready like ready_message,       /* Ready msg information */
      2 warning like ready_message,     /* Warning msg information */
      2 error like ready_message,       /* Same for errors */
      2 static_on_units (10),           /* list of 10 sous*/
         3 sou_ecb ptr options(short), /* ptr to ecb*/
         3 sou_status fixed bin(15),    /* sou_status and cntr*/
      2 search_list_ptr ptr options(short), /* search list head ptr */
      2 smt_list_ptr ptr options(short),/*ptr to list of active EPFs*/
      2 epf_cache_hd_ptr ptr options(short), /* EPF cache head ptr */
      2 epf_cache_tl_ptr ptr options(short), /* EPF cache tail ptr */
      2 epf_cache_count fixed bin(15),       /* EPF cache counter */
                           ~~
                           ~~

/* The first stack frame beginneth here. */

      2 first_fr fixed bin(15);
```

STANDARD COMMAND PROCESSOR STD$CP



Section 8 - Executable Program Format Files

Objectives: The student will be able to

- o name the data structures created by BIND
- o describe the phases in the life of an EPF
- o explain how the BIND-created data structures are used in EPF startup
- o explain the data structures built by PRIMOS to manage EPFs

STATIC VS DYNAMIC RUNFILES

STATIC	DYNAMIC
.SEG, .SAVE	.RUN
SEG or LOAD loaders	BIND loader
Uses the same static segments for every invocation as assigned by SEG/LOAD	Uses available dynamic segments for every invocation as assigned by PRIMOS
Contains virtual addresses	Contain EPF Relocatable Pointers ERPs
Contains procedure and linkage images	Contains procedure image and a description of the linkage area(s)
Entire runfile is read into memory and paging space allocated	Procedure images mapped to memory via VMFA, required linkage is built, and paging space allocated for linkage; procedure read into memory as needed
User manages address space	PRIMOS manages address space
Limited restartability of command environment	Full restartability of command environment
Uses private stack (4xxx)	Uses command processor stack
Must be explicitly shared	Are implicitly shared

EXECUTABLE PROGRAM FORMAT - EPF

The Executable Program Format (EPF) implements a new program object representation for V-mode programs. EPFs, unlike static mode runfiles which have their virtual addresses assigned by the loader/linker, are not associated with virtual addresses until runtime. Therefore, the format of a .RUN file as well as the steps taken to execute it greatly differ from its static mode counterpart.

```
                          +---------------------------+
1. VCIB                   |EPF identifier             |
                          |size of this file          |
                          |size of linkage to build   |
                          |ERP (rel ptr) to CIB       |
                          +---------------------------+
2. PROCEDURE IMAGE        |procedure image 1          |
                          |         .                 |
                          |procedure image n          |
                          +---------------------------+
3. CIB                    |ERPs to rest of the EPF    |
                          |file structure :           |
                          | * linkage description     |
                          | * library info block      |
                          | * DBG info block          |
                          | * misc. info blocks       |
                          +---------------------------+
4. LINKAGE DESCRIPTION    |LTD1 --> lte list          |
                          |     --> dtb list          |
                          |  .                        |
                          |  .                        |
                          |  .                        |
                          |LTDn --> lte list          |
                          |     --> dtb list          |
                          +---------------------------+
5. LIBRARY INFORMATION    |search type                |
                          |size of table              |
                          |ent pt table ptr           |
                          |# entries in table         |
                          +---------------------------+
6. MISC. INFO             |command line options       |
                          |comments                   |
                          |etc.                       |
                          +---------------------------+
7. DBG INFO               |                           |
                          +---------------------------+
```

EPF LOGICAL STRUCTURE

```
                         ---------  ---------
                         |  lte  |  |  lte  |
          ----------->   ---------  ---------

          ----------->   ---------  ---------
                         |  dtb  |  |  dtb  |
                         ---------  ---------


                         ---------
                         |  lte  |
            --------->   ---------

            --------->   ---------
                         |  dtb  |
                         ---------
```

THE VERY CRITICAL INFORMATION BLOCK - VCIB

The information stored by BIND in the VCIB is critical to PRIMOS in the initial phase of EPF invocation. Hence, the VCIB comes first in the EPF runfile.

```
1               7 8             16
 ________________________________
|    STARTING ADDRESS            |   ----> always -1 for an EPF
 ________________________________
|    ENDING ADDRESS              |   ----> always 0 for an EPF
 ________________________________
|   TYPE        |   VERSION #    |   ----> types:
 ________________________________              1 = prog_always_reinit
| # SEGS NEEDED FOR RESUME       |             3 = process_class_library
 ________________________________              4 = program_class_library
|   # OF LINKAGE AREAS           |
 ________________________________
| # SEGS NEEDED FOR DBG          |
 ________________________________
|           CIB                  |
|   -   -   -   -   -   -        |
|           ERP                  |
 ________________________________
```

THE CRITICAL INFORMATION BLOCK - CIB

The information stored in the CIB by BIND allows PRIMOS to access the many elements, such as the starting ECB and the linkage descriptors contained within the EPF runfile. The CIB is accessed during the various phases of EPF startup.

```
 ________________________________
|        VERSION OF CIB          |
|________________________________|
|           STARTING             |
| -  -  -  -  -  -  -  -  -  -   |
|           ECB ERP              |
|________________________________|
|            ERP TO              |
| -  -  -  -  -  -  -  -  -  -   |
|           LTDS LIST            |
|________________________________|
|         LIBRARY ENTRY          |
| -  -  -  -  -  -  -  -  -  -   |
|       POINT TABLE ERP          |
|________________________________|
|                                |
~~                              ~~
|                                |
|________________________________|
|        DBG INFORMATION         |
| -  -  -  -  -  -  -  -  -  -   |
|              ERP               |
|________________________________|
|                                |
~~                              ~~
|                                |
|________________________________|
|  CMD PROC FEATURES FLAGS       |
|________________________________|
|                                |
~~                              ~~
|                                |
|________________________________|
|    ADDITIONAL INFORMATION      |
| -  -  -  -  -  -  -  -  -  -   |
|              ERP               |
|________________________________|
```

THE LINKAGE DESCRIPTION

The linkage area(s) of an EPF are constructed at runtime from a ´description´ created by BIND. The description consists of three types of data structures: LTDs, LTEs, and DTBs.

Linkage Template Descriptor (LTD)

- o Describes a linkage area
- o Contains the following information:
  - o Size of the linkage area
  - o ERP to its list of LTEs
  - o ERP to its list of DTBs

Linkage Template Entry (LTE)

- o Describes one type of data
- o Types of data include:
  - o ECBs
  - o IPs
  - o Faulted IPs
  - o Static data
  - o Repeated data

Data Template Block (DTB)

- o Contains the actual data described by a corresponding LTE

THE LIFE OF AN EPF

The life of an EPF can be viewed in phases:

- o Mapping the procedure segment(s) to memory (EPF$MAP).
- o Allocating the necessary memory for linkage (EPF$ALLC).
- o Initializing the linkage area(s) and relocating all addresses (EPF$INIT).
- o Invoking the EPF (EPF$INVK).
- o Deleting the EPF from memory (EPF$DEL).

THE ACTIVE SEGMENT TABLE - AST

In order to keep track of the EPF procedure segments currently in memory, PRIMOS maintains the Active Segment Table (AST). The AST consists of entries (ASTEs), one for each EPF procedure segment currently in memory. The number of ASTEs is determined by the setting of the config directive, NVMFS. The AST resides in segment 14. Following is the format of an ASTE.

1 7	8 16
ADDRESS OF PAGE MAP	
DEVICE NUMBER	HI-ORD 8B BRA
LOW ORDER 16 BITS BRA	
PREV RA	NEXT RA
LOW 16 BITS - PREDECESSOR RA	
LOW 16 BITS - SUCCESSOR RA	
# WINDOW INTO VMFA FILE	
# ACTIVE PAGES IN SEGMENT	
# READERS	# WRITERS
CONCURRENCY	

THE EPF MAPPING PHASE - EPF$MAP.PLP

When an EPF is RESUMEd, EPF$MAP calls VINIT$ to map each of the procedure segments of the EPF into memory. EPF$MAP must access the VCIB, which resides in the first procedure segment in order to tell VINIT$ how many more procedure segments are to be mapped into memory.

For each procedure segment, VINIT$ performs the following steps:

- o If the procedure segment is already mapped into memory for a process other than the requesting process, VINIT$ finds an unused dynamic segment (i.e., SDW) in the process´ DTAR2, increments the ASTE readers count, and returns the segment number to EPF$MAP.

- o If the procedure segment is already mapped into memory for the requesting process, VINIT$ returns the number of the segment that´s already mapped into the user´s address space to EPF$MAP.

- o If the procedure segment is not mapped into memory at all, VINIT$ finds an unused dynamic segment (i.e. SDW) in the process´ DTAR2, initializes a new ASTE, and returns the segment number to EPF$MAP.

- o If the EPF is on a remote disk, VINIT$ finds a free dynamic segment (i.e., SDW) in the process´ DTAR2 , calls PRWF$$ to copy the data into the segment, and returns its number to EPF$MAP. That is, it is not handled like a local EPF.

THE SEGMENT MAPPING TABLE - SMT

Each process using an EPF must keep track of the status and virtual mapping for its use of that EPF. The table dynamically created by EPF$MAP is called a Segment Mapping Table (SMT). There is one SMT for each EPF that a process has mapped into memory, and they are linked together (head of list pointer in CLDATA). There are four pieces to the SMT:

- SMT.STABLE_ENT contains information about the EPF, derived from both the VCIB and the CIB, that will not change regardless of the number of invocations.
- SMT.ACTIVE_ENT contains the volatile information including the current status.
- SMT.SEGS(n) is the SMT address table that keeps track of the virtual addresses assigned to this invocation of the EPF.
- SMT.EPF_PATHNAME contains the character count and full pathname of the EPF.

SMT FORMAT

```
                      ________________________________
STABLE ENTRY -->     |# procedure segments            |
                      ________________________________
                     |# linkage area                  |
                      ________________________________
                     |origin ptr (2 words)            |
                      ________________________________
                     |EPF pathname ptr (2 words)      |
                      ________________________________
                     |next SMT ptr                    |
                      ________________________________
                     |lib.search_type                 |
                      ________________________________
                     |lib.ent_tbl ptr (2 words)       |
                      ________________________________
                     |lib.ent_tbl_size                |
                      ________________________________
                     |lib.ent_num                     |
                      ________________________________
                     |lib.link_ref_ctr (2 words)      |
                      ________________________________
                     |epf type     | epf version      |
                      ________________________________
                     |flags : dbg,cache,init ...      |
                      ________________________________
ACTIVE ENTRY -->     |command level                   |
                      ________________________________
                     |flags: link init and alloc      |
                      ________________________________
                     |prev act_ent ptr (2 wrds)       |
                      ________________________________
SMT ADDR TABLE -->   |seg no. for last linkage        |
                      ________________________________
                     .                                .
                     .                                .
                      ________________________________
ORIGIN PTR -->       |seg no. for first proc          |
                      ________________________________
                     .                                .
                     .                                .
                      ________________________________
EPF PATHNAME -->     |length of pathname              |
                      ________________________________
                     |pathname .......                |
                      ________________________________
                     .                                .
                     .                                .
                     .                                .
                      ________________________________
```

SMT ADDRESS TABLE

The SMT address table keeps track of the virtual addresses that are assigned to the EPF procedure segments and linkage areas. Each entry will eventually hold the 2-word virtual address assigned to that procedure segment or linkage area to be used as the base address for the relocation of ERPs. The index into the table is the relative segment number portion of an ERP. A sample address table is shown below.

```
                                    ______________________________________
SMT.ACTIVE_ENT.SEGS --> -n*2       |         relocation address           |
                                   | - - - - - - - - - - -                |
                                   |         for nth linkage area         |
                                    ______________________________________
                                   |                                      |
                                   ~                                      ~
                                   ~                                      ~
                                   |                                      |
                                    ______________________________________
                             -2    |         relocation address           |
                                   | - - - - - - - - - - -                |
                                   |       for first linkage area         |
                                    ______________________________________
SMT.STABLE_ENT.ORIGIN --> 0        |         relocation address           |
                                   | - - - - - - - - - - -                |
                                   |   for first procedure segment        |
                                    ______________________________________
                              2    |         relocation address           |
                                   | - - - - - - - - - - -                |
                                   |   for second procedure segment       |
                                    ______________________________________
                              4    |         relocation address           |
                                   | - - - - - - - - - - -                |
                                   |   for third procedure segment        |
                                    ______________________________________
                                   ~                                      ~
                                   ~                                      ~
```

THE ALLOCATION PHASE - EPF$ALLC.PLP

Once the procedure image is mapped into memory via VMFA, the memory for the linkage area(s) can be allocated and their address(es) stored off in the SMT address table. In order to perform the linkage allocation phase, EPF$ALLC examines the LTDs for the sizes of the linkage areas.

```
          CIB
 ________________
|                |
|                |                     LTDs
|----------------|
| LTD LIST ERP   |-----------> ______________
|----------------|             |            |
|                |             |------------|
|                |             |    SIZE    |
|                |             |------------|
|                |             |____________|
|________________|             |            |
                               |------------|
                               |    SIZE    |
                               |------------|
                               |____________|
                                     ~~
```

THE INITIALIZATION PHASE - EPF$INIT.PLP

Once the linkage has been allocated, EPF$INIT performs the initialization phase. The following table lists the types of data and the initialization steps.

DATA TYPE	ACTIONS
STATIC	COPIED FROM DTB
UNINITIALIZED	NO ACTION
REPEATED	COPIED FROM DTB & EXPANDED
ECB(S)	COPIED FROM DTB & RELOCATE PB, LB
INDIRECT POINTERS	RELOCATE
FAULTED INDIRECT POINTERS	RELOCATE & SET FAULT BIT
STATIC INDIRECT POINTERS	COPIED FROM DTB NOT RELOCATED

THE INVOCATION PHASE - EPF$INVK.PLP

To invoke the EPF, EPF$INVK creates an EPF cache entry and inserts it at the head of the process' cache list, and then calls the EPF. When the EPF returns, its cache entry is left threaded onto the cache list, but its SMT is marked as being inactive. Another invocation of the EPF, while its cache entry is still threaded on the cache list, will only have to go through a partial initialization (i.e., static data and faulted IPs) of the linkage area.

An EPF's cache entry will remain on the cache list until it is removed because

(1) the cache list has become full, and it is the least recently used entry,
(2) it has been explicitly removed with the Remove_Epf command,
(3) the user's ring 3 environment has been reinitialized, or
(4) a new command level is pushed (see next page).

Removal of a cache entry will cause EPF$DEL to be called to remove the SMT from the process' SMT list (CLDATA.SMT_LIST_PTR) and delete the SMT from memory. A subsequent invocation of the EPF must then go through all phases.

```
CLDATA.EPF_CACHE_HD_PR                               CLDATA.EPF_CACHE_TL_PTR
|                                                    |
|                                                    |
v                                                    v
---------------     |---  ---------------     |---   ---------------
| A(NEXT ENT) | <--       | A(NEXT ENT) | <---       | A(NEXT ENT) |
---------------           ---------------            ---------------
| A(PREV ENT) |           | A(PREV ENT) |            | A(PREV ENT) |--> null
---------------           ---------------            ---------------
|   A(SMT)    |           |   A(SMT)    |            |   A(SMT)    |
---------------           ---------------            ---------------
```

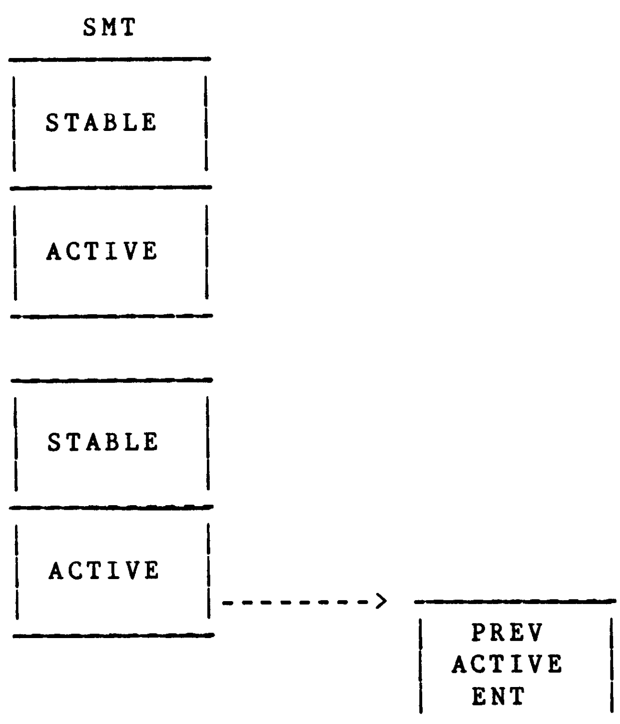
MOVING BETWEEN COMMAND LEVELS

If an EPF is broken out of (i.e., ^P was typed during execution), a new command level is pushed. Before the new command level is initialized, the previous command level is ´cleaned up´. Cache entries in the previous command level cache list representing inactive EPFs are popped from the list. Hence, only active EPFs are ´carried forward´ to the new command level.

If an already active EPF were to be reinvoked at the new command level, the linkage area assignments for both the original invocation and the new invocation must be preserved. To ensure this, a copy (in the diagram called PREV_ACTIVE_ENT) is made of the SMT.ACTIVE_ENT for the original invocation. SMT.ACTIVE_ENT is then initialized to show that a new command has been pushed, and the addresses for the linkage areas are set to null. These addresses will then be filled in upon reinvocation of that EPF.

COMMAND LEVEL X:

COMMAND LEVEL X+1:



Section 9 - File System

Objectives: The student will be able to

- o describe physical disk data structure formats
- o describe the various file types and their advantages
- o describe ACL data structures
- o explain how the LOCATE mechanism works
- o describe unit table data structures

PHYSICAL DISK STRUCTURES

A disk drive is divided into one or more partitions where a partition is one or more pairs of heads. Each partition must contain:

1). MFD (Master file directory)

2). DSKRAT (Disk record availability table)

3). BOOT (For initial loading)

4). UFD DOS (Initially empty - not actually required)

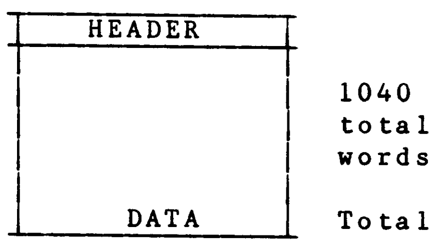
5). UFD CMDNCO (Initially empty)

6). BADSPT (If badspots on the disk)

Each partition is divided into 1040 word records.

The record header is 16 words for storage modules devices.

The remainder of the record holds data (1024 words).



RECORD HEADER FORMAT - 1040 WORD

0	REKCRA	RECORD ADDRESS OF THIS RECORD
1		
2	REKPOP	RA OF DIRECTORY ENTRY OF THIS RECORD
3		
4	REKDCT	NUMBER OF DATA WORDS IN RECORD
5	REKTYP	TYPE OF FILE (Only on first record)
6	REKFPT	RA OF NEXT SEQUENTIAL RECORD
7		
8	REKBPT	RA OF PREVIOUS RECORD
9		
10	REKLVL	INDEX LEVEL FOR DAM FILES
11	Reserved	
12		
13		
14		
15		

DSKRAT FORMAT

```
dcl 1 disk_rat based,               /* Usually found in LOCATE buffer */
     2 len fixed bin,               /* no. of words in DSKRAT header  */
     2 rec_size fixed bin,          /* phys. record size (448 or 1040)*/
     2 disk_size fixed bin(31),     /* number of records in partition */
     2 heads fixed bin,             /* number of heads in partition   */
     2 spec_bits,
        3 dummy bit(14),
        3 crash bit(1),             /* improperly shut down last time */
        3 dos bit(1),               /* DOS modified or perm. broken   */
     2 cyls fixed bin,              /* number of cylinders (tracks)   */
     2 rev_num fixed bin,           /* Rev. number */
     2 rat(0:1015) bit (16) aligned; /* The RAT itself */
```

BADSPOT FILE FORMAT - Data Structures

- BADSPT file header:

```
dcl 1 badspt_file_header,
      2 bad_blk_off fixed bin,  /* offset of the 1st badspt blk */
      2 MBZ fixed bin,          /* must be zero                 */
      2 file_size fixed bin,    /* size of the badspt file      */
      2 reserve(5) fixed bin;
```

- Badspot entry:

```
dcl 1 badspt_blk_header,
      2 bcw,                 /* block control word                     */
        3 type bit(4),       /* block type (badspt blk type = 0)       */
        3 length bit(12),    /* length of this block                   */
      2 badspt_blk((badspt_blk_header.bcw.length-1)/2)
        3 track fixed bin,   /* track number                           */
        3 sector bit(8),     /* sector number+1, 0 for whole track*/
        3 head bit(8);       /* head number                            */
```

- Remapped badspot entry:

```
dcl 1 eqv_blk_header,
      2 bcw,                     /* block control word         */
        3 type bit(4),           /* type of this block
                                    (eqv blk type = 1)         */
        3 length bit(12),        /* length of this block       */
      2 eqv_blk((eqv_blk_header.bcw.length-1)/2)
        3 bad_track fixed bin,   /* bad track number           */
        3 bad_sector bit(8),     /* bad sector number+1        */
        3 bad_head bit(8),       /* bad head number            */
        3 eqv_track fixed bin,   /* equivlant track number     */
        3 eqv_sector bit(8),     /* equivlant sector number+1 */
        3 eqv_head bit(8);       /* equivlant head number      */
```

DIRECTORY STRUCTURE

- A directory is a header followed by a bunch of entries.

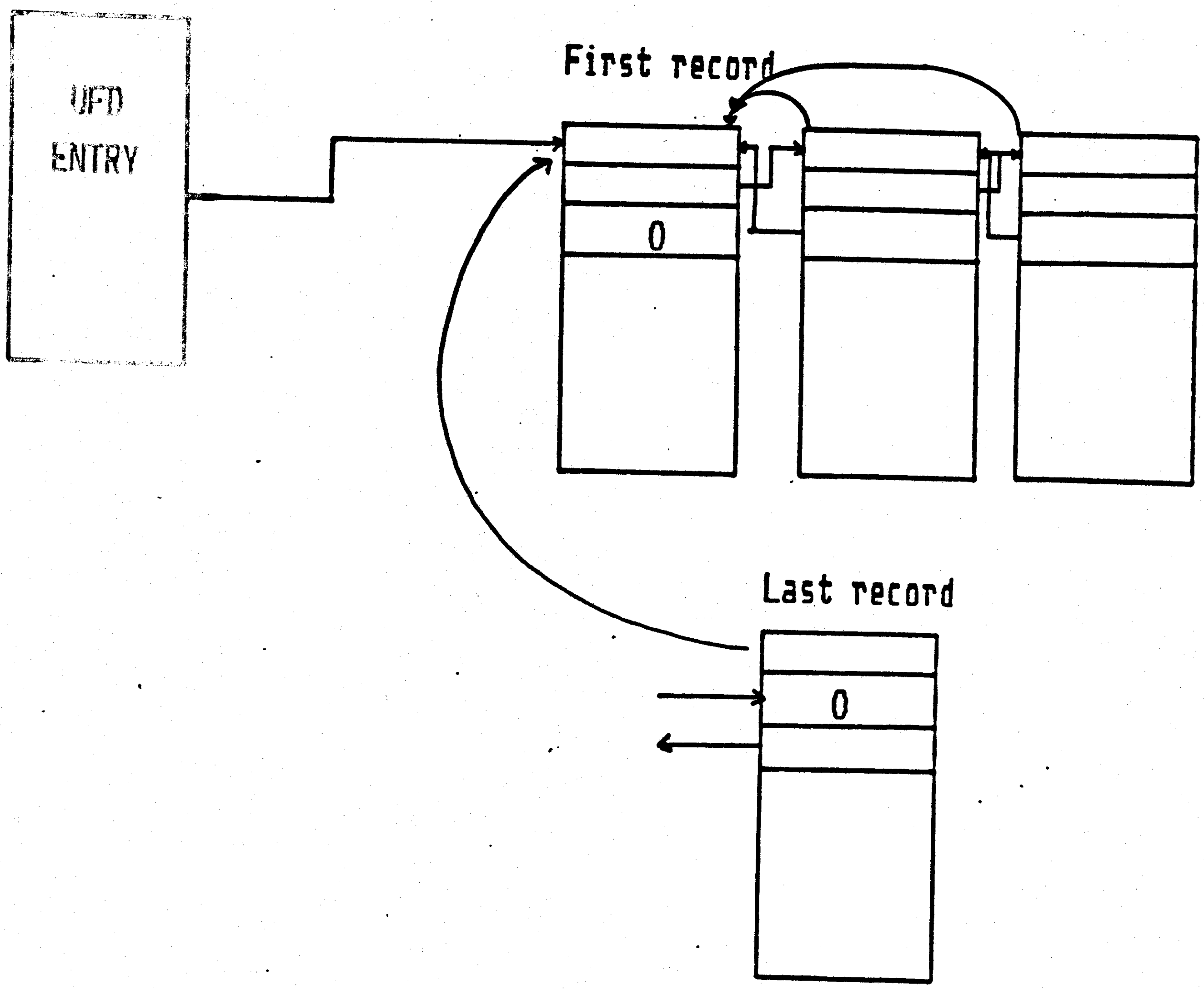
Directory Header
File Entry
ACL
hole
Directory Entry

- Note, ACLs are embedded in the directory itself.
A UFD is a SAM file. Max size is <=64KW.

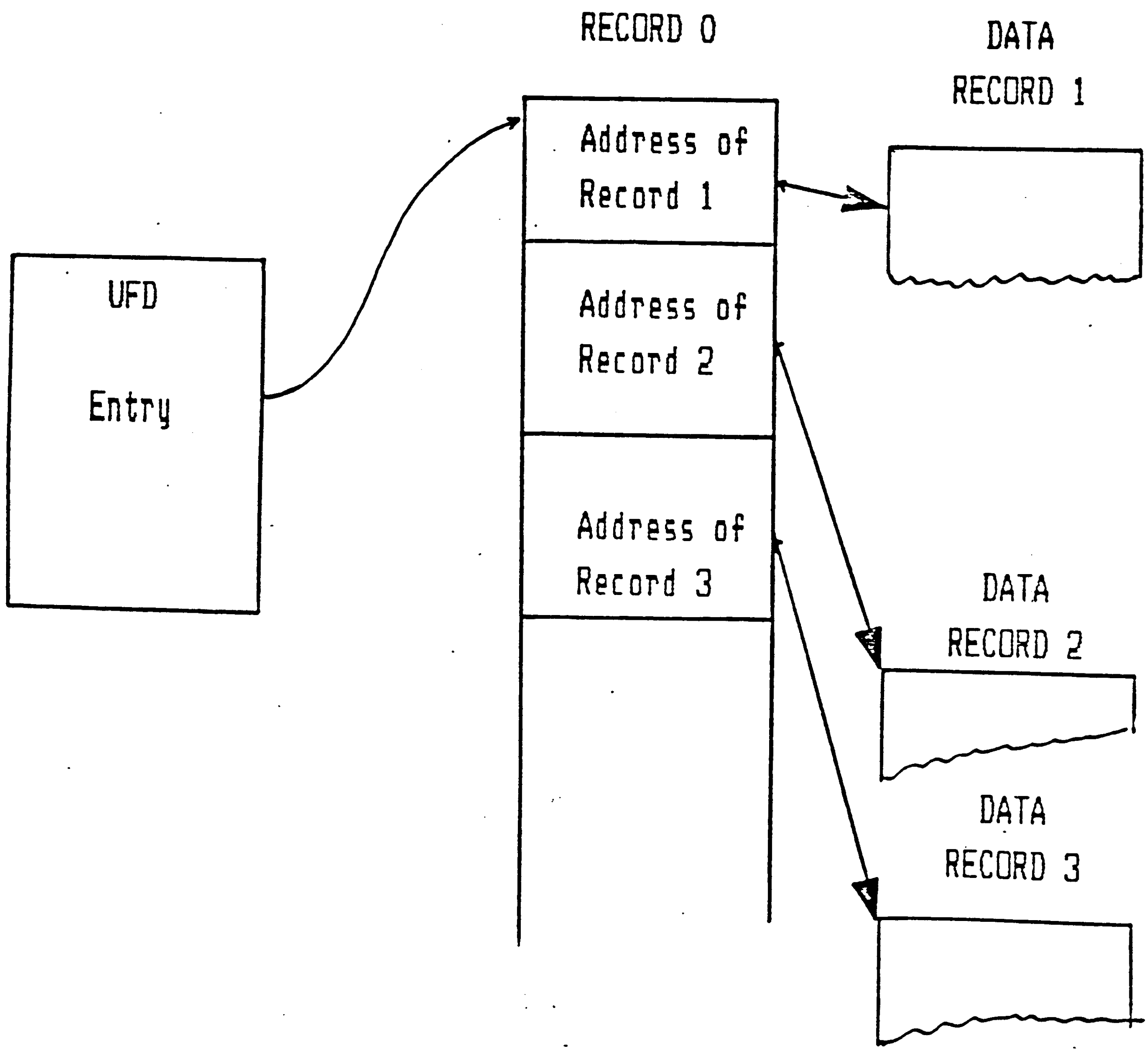
DIRECTORY STRUCTURE

```
dcl 1 dir_hdr based,                       /* dir header entry structure */
      2 ecw like ecw,
      2 owner_password char(6),            /* Owner password             */
      2 non_owner_password char(6),        /* Nonowner password          */
      2 sparel fixed bin,
      2 max_quota fixed bin (31),          /* Max Quota                  */
      2 dir_used fixed bin (31),           /* Quota used in this dir     */
      2 tree_used fixed bin (31),          /* Quota used in whole subtree*/
      2 rec_time_prod fixed bin (31),      /* Record/time product        */
      2 prod_dtm like fsdate,              /* DTM of record/time product */
      2 spare2(5) fixed bin;

dcl 1 ecw based,                           /* Entry control word         */
      2 type bit(8),                       /* Type of entry              */
      2 len bit(8);                        /* Length of entry            */

replace dir_hdr_ecwt by '01'b4,            /* ECW types: directory header*/
        vacant_ecwt  by '02'b4,            /*            vacant entry    */
        file_ecwt    by '03'b4,            /*            file entry      */
        acc_cat_ecwt by '04'b4,            /*            access category */
        acl_ecwt     by '05'b4;            /*            ACL itself      */
```

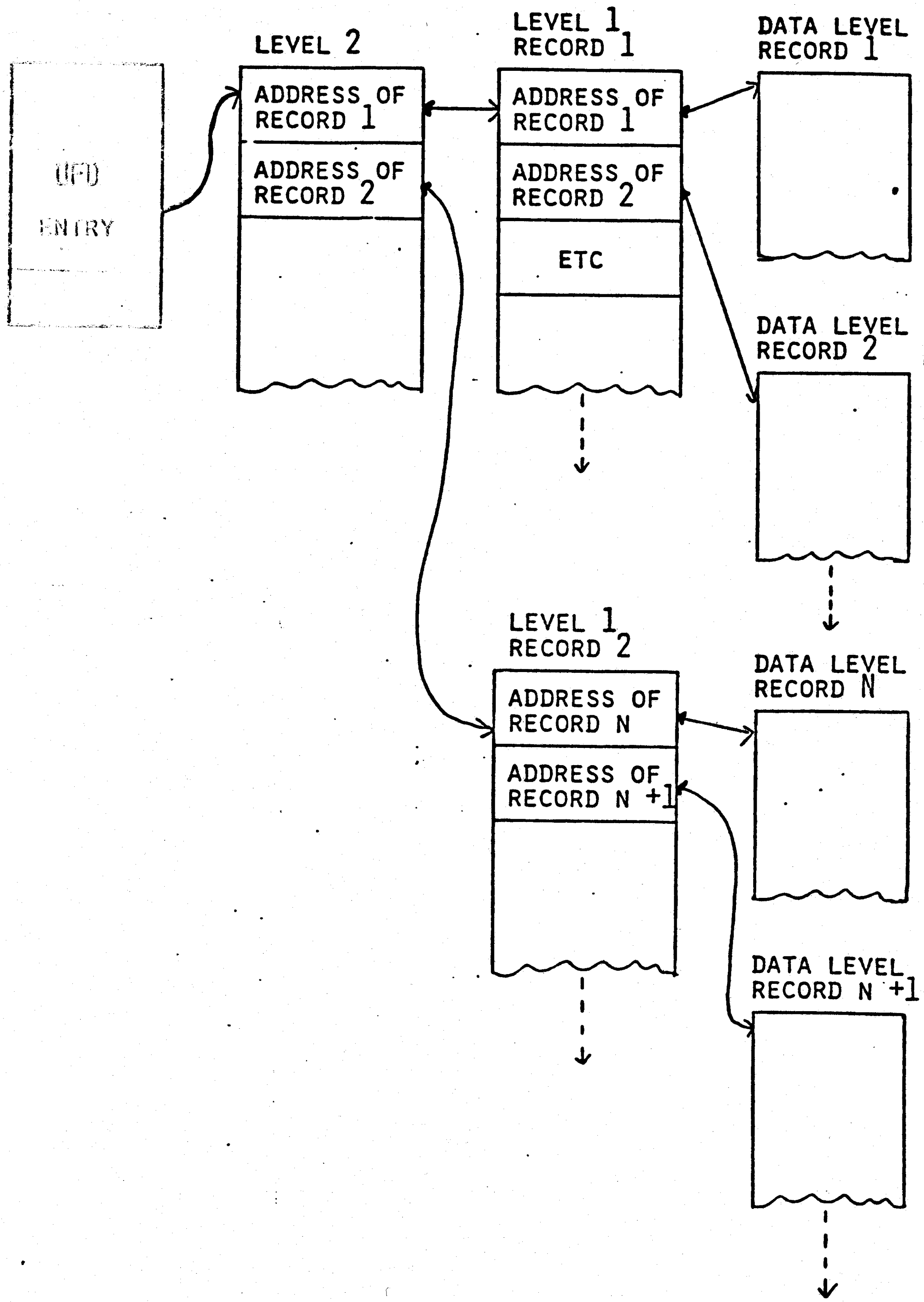
SAM FILES



DAM FILES


RECORD 0
DATA
RECORD 1
UFD
Entry
Address of
Record 1
Address of
Record 2
Address of
Record 3
DATA
RECORD 2
DATA
RECORD 3

MULTILEVEL DAM FILES



SEGMENT DIRECTORY FORMAT

0	BRA 0	Beginning record address
1		of first file in directory
2	BRA 1	Beginning record address
3		of second file in directory
4	0	Null entry
5		
2n	BRA n	Beginning record address
2n+1		of last file in directory

DIRECTORY STRUCTURE - NORMAL ENTRY

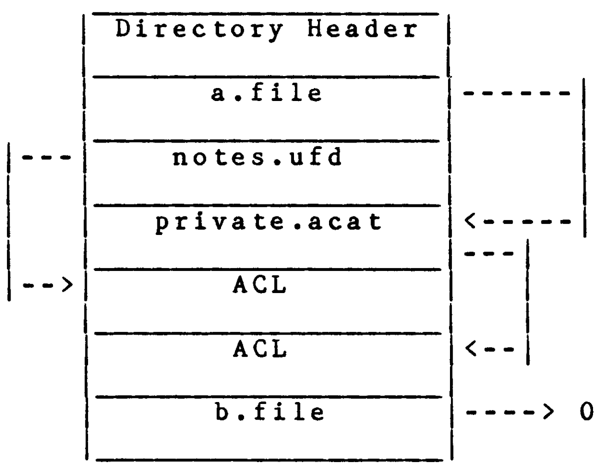
- Normal entry for a file or directory:

```
dcl 1 file_ent based,               /* Structure of file entry        */
      2 ecw like ecw,
      2 bra fixed bin (31),         /* bra of file                    */
      2 log_type fixed bin,         /* logical type attribute         */
      2 dtb like fsdate,            /* Date/time last backed up       */
      2 protec bit (16),            /* Protection keys                */
      2 acl_pos fixed bin,          /* Position of ACL, assumes
                                       dir <= 64k                     */
      2 dtm like fsdate,
      2 file_info,
        3 long_rat_hdr bit (1),     /* '8000'b4:  file is a long RAT */
        3 dumped bit (1),           /* '4000'b4:  has been backed up */
        3 dos_mod bit (1),          /* '2000'b4:  modified under DOS */
        3 special bit (1),          /* '1000'b4:  Special file       */
        3 rwlock bit (2),           /* Bits 5-6:  Concurrency lock   */
        3 trunc bit (1),            /* Bit 7:  truncated by FIX_DISK */
        3 spare bit (1),            /* Bit 8:     Unused             */
        3 type bit (8),             /* Bits 9-16: File type          */
      2 scw fixed bin,              /* Length of name subentry       */
      2 name char (32);             /* Name of object                */
```

DIRECTORY STRUCTURE - ACL POSITION

- ACL_POS

  Position in the directory of the ACL protecting this object.
  if specific protection then pointer is to an ACL.
  if category protection then pointer is to access category.
  if default protection then pointer is zero.



- Note, the ACL protecting this directory lives in the parent directory along with the entry describing this directory.

DIRECTORY STRUCTURE - ACL ENTRY

- Directory entry for an ACL:

```
dcl 1 acl_ent based,                  /* Dir entry for an ACL          */
      2 ecw like ecw,                 /* See above                     */
      2 user_count fixed bin,         /* Number of user entries        */
      2 group_count fixed bin,        /* Number of group entries       */
      2 version fixed bin,            /* Version number of structure */
      2 spare1 fixed bin,
      2 group_offset fixed bin,       /* Relative position of first
                                         group entry                   */
      2 rest_accesses like accesses,  /* Rights for $REST              */
      2 owner_pos fixed bin,          /* Position of owner in dir      */
      2 dtm like fsdate,              /* Date/time last modified       */
      2 spare2 fixed bin,
      2 entry like coded_access;      /* See below */
```

- Format of a single access pair:

```
dcl 1 coded_access based,       /* Entry in an ACL        */
      2 scw fixed bin,          /* Length only            */
      2 access like accesses,   /* <access>               */
      2 spare(2) fixed bin,
      2 id char(32) var;        /* <id> */

dcl 1 accesses based,           /* A 16-bit access word */
      2 ring1 like acc_bits,
      2 ring3 like acc_bits;

dcl 1 acc_bits based,           /* Access bit definition           */
      2 protect bit(1),         /* Directory accesses -- Protect */
      2 delete bit(1),                               /* Delete  */
      2 add bit(1),                                  /* Add     */
      2 list bit(1),                                 /* List    */
      2 use bit(1),                                  /* Use     */
      2 execute bit(1),         /* File accesses --     Execute */
      2 write bit(1),                                /* Write   */
      2 read bit(1);                                 /* Read    */
```

DIRECTORY STRUCTURE - ACCESS CATEGORY ENTRY

- An access category is a named ACL.

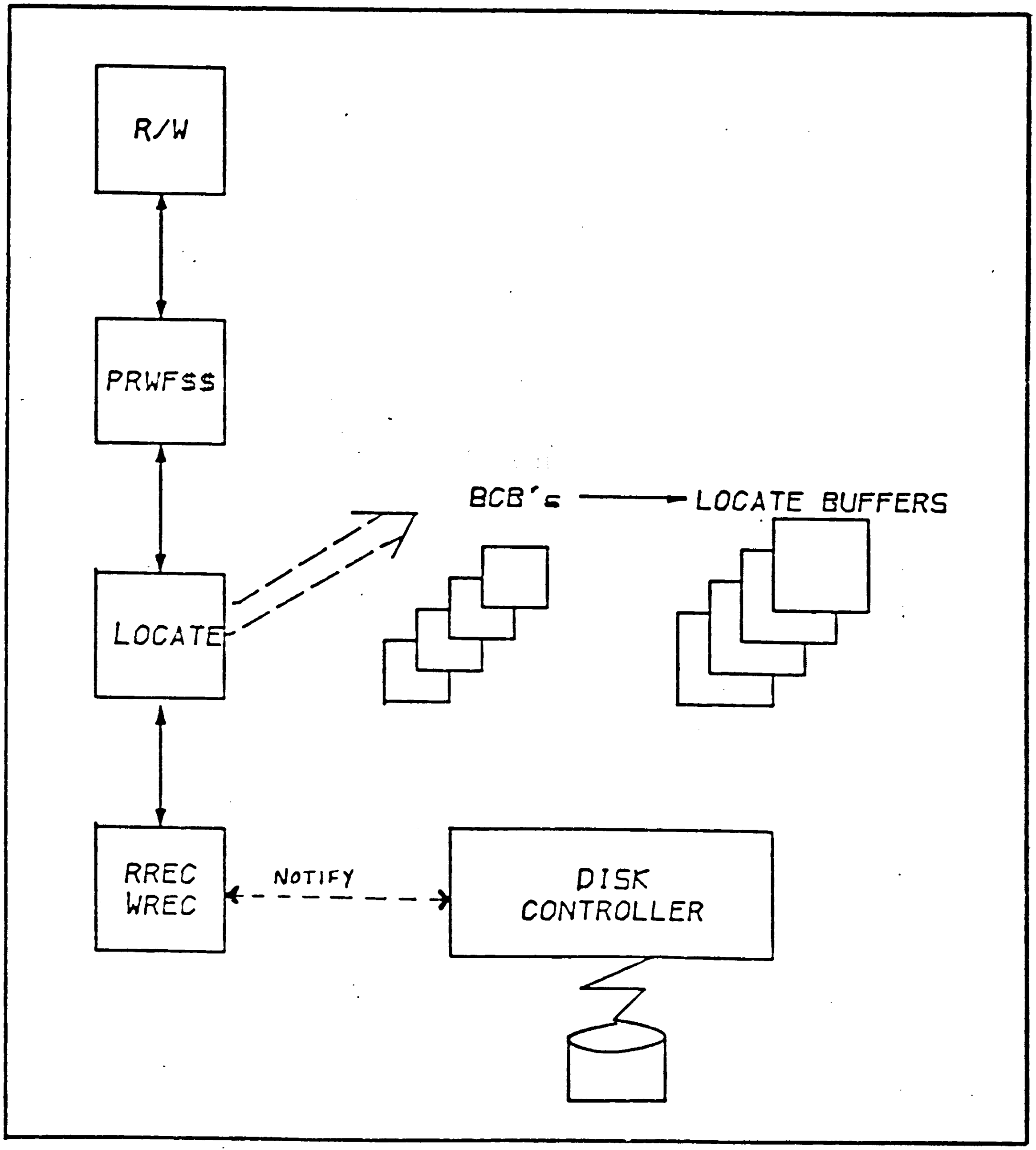
- It is a pointer to an ACL entry.

- Each file system object protected by the category points to the access category entry, not the ACL itself.

- The name field of an access category is always padded to 32 characters in order to reduce directory fragmentation.

```
dcl 1 acc_cat_ent based,     /* access category directory entry     */
      2 ecw like ecw,
      2 spare1(3) fixed bin,
      2 dtls like fsdate,    /* Date/time last saved                */
      2 spare2(1) fixed bin,
      2 acl_pos fixed bin,   /* Position of ACL itself              */
      2 dtm like fsdate,     /* Date/time last modified             */
      2 file_type fixed bin, /* For compatibility with normal entry */
      2 scw fixed bin,       /* Length of name subentry             */
      2 name char (32);      /* Name of object (padded to 32 chars) */
```

THE LOCATE MECHANISM



BUFFER CONTROL BLOCK

Word	Contents		Name	
0	HASH THREAD		BUFLNK	
1	Logical dev	Record	BUFRA	
2	ADDRESS			
3	BRA of file record is in		BUFBRA	
4				
5	Process no.	Hash index	BUFUSR	
6	User count	Flag bits	BUFLAG	
7			REKCRA	
10				
11			REKPOP	
12				
13			REKDCT	
14			REKTYP	disk
15			REKFPT	record
16				header
17			REKBPT	
20				
21			REKLVL	
22	ADDRESS OF PTW		BUFPMP	
23	FOR BUFFER			
24	LRU THREAD FOR		BUFTHD	
25	UNUSED BUFFERS			

FLAG BITS 16 = BUFFER MODIFIED
15 = BUFFER IN TRANSITION
14 = UPDATE MISSED

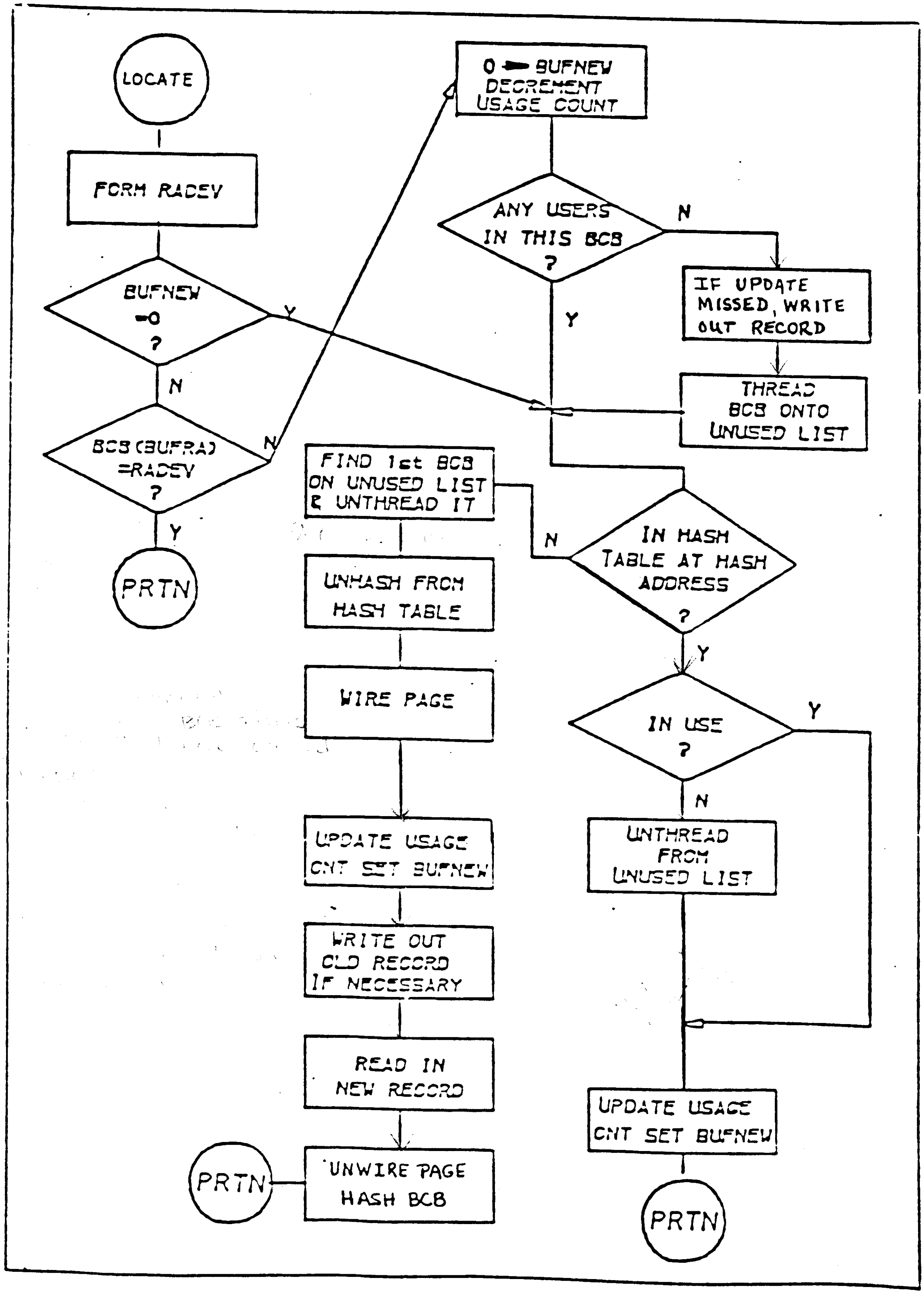
MANAGING BCBs

BCBs are: on the "unused list"
in the Hash Table
or in both places

A chart:

on the Unused list? \ In the Hash Table?	no	yes
no		
yes		

LOCATE.PMA


LOCATE
FORM RADEV
BUFNEW =0 ?
Y
N
BCB (BUFRA) =RADEV ?
N
Y
PRTN
0 → BUFNEW DECREMENT USAGE COUNT
ANY USERS IN THIS BCB ?
N
Y
IF UPDATE MISSED, WRITE OUT RECORD
THREAD BCB ONTO UNUSED LIST
FIND 1st BCB ON UNUSED LIST & UNTHREAD IT
N
IN HASH TABLE AT HASH ADDRESS ?
Y
UNHASH FROM HASH TABLE
WIRE PAGE
IN USE ?
Y
N
UPDATE USAGE CNT SET BUFNEW
UNTHREAD FROM UNUSED LIST
WRITE OUT OLD RECORD IF NECESSARY
READ IN NEW RECORD
UPDATE USAGE CNT SET BUFNEW
PRTN
UNWIRE PAGE HASH BCB
PRTN

ASSOCIATIVE BUFFERS - CONFIG DIRECTIVE

Previously- there were always 64 associative buffers which resided in segment 1.

Now there can be any where from 8 to 256 associative buffers.

New CONFIG directive: NLBUF n
where n = the octal number of LOCATE buffers to use.

The buffers will reside in segments 50 - 53.

The 22 word Buffer Control Block (BCB) is wired at cold start.
The LOCATE buffer is only wired when it is in use.

The optimal number of associative buffers depends on the system.
If the LOCATE miss rate is greater than 10 percent,
NLBUF should be increased until %MISS is less than 10%
However, if PF/S is greater than 10, do not increase NLBUF.

Be sure that LM/S is high enough to make %MISS meaningful.

UNIT TABLES - Definitions

- A unit table (ut) is a list of pointers to unit table entries.

- A hash table is a set of pointers to linked lists of unit table entries.

- A unit table entry (ute) desribes a file system object that is currently in use via the file system.

- A file system object is a data file, directory or access category. These objects may reside on a local or a remote system.

UNIT TABLES

OLD METHOD

- Per-User unit tables allocated/deallocated dynamically.
- Constrains working set of unit table databases to what is actually being used.
- Vital statistics:

  3247 file units available per system

  8 guaranteed per user (default)
  1 system unit per user (unit #0)
  3 attach points (home,current,initial) per user
  127 maximum ´usable´ file units per user

NEW METHOD

- Per-user unit tables allocated/deallocated dynamically.
- Maximum of 32768 units per user.
- Unit table dynamically grows as more file units are requested.
- Initially, get 38 file units:

  -5 temporary attach
  -4 como
  -3 IAP
  -2 home
  -1 current
  0 system
  1-32 available for user

UNIT TABLE

```
pudcom.utblptr --->  -----------------------------
                     | current max file unit no. |
                     -----------------------------
                     |           rfu             |
                     -----------------------------
                     |     temporary attach      |
                     |  -  -  -  -  -  -  -  -   |
                     |        UTE pointer        |
                     -----------------------------
                     |          como             |
                     |  -  -  -  -  -  -  -  -   |
                     |        UTE pointer        |
                     -----------------------------
                     |   initial attach point    |
                     |  -  -  -  -  -  -  -  -   |
                     |        UTE pointer        |
                     -----------------------------
                     |    home attach point      |
                     |  -  -  -  -  -  -  -  -   |
                     |        UTE pointer        |
                     -----------------------------
                     |   current attach point    |
                     |  -  -  -  -  -  -  -  -   |
                     |        UTE pointer        |
                     -----------------------------
                     |         system            |
                     |  -  -  -  -  -  -  -  -   |
                     |        UTE pointer        |
                     -----------------------------
                     |       file unit 1         |
                     |  -  -  -  -  -  -  -  -   |
                     |        UTE pointer        |
                     -----------------------------
                     ~~                         ~~

                     ~~                         ~~
                     -----------------------------
                     |       file unit 32        |
                     |  -  -  -  -  -  -  -  -   |
                     |        UTE pointer        |
                     -----------------------------
```

A NON-ATTACH POINT UTE

```
Dcl 1 utcme based,                /* File/Directory Unit Table Entry */
      2 vstat like status_bits,      /* See below                       */
      2 bra fixed bin (31),          /* BRA of file                     */
      2 cur_ra fixed bin (31),       /* current r.a. in file            */
      2 ldevno fixed bin,            /* logical device number           */
      2 rel_wordno fixed bin,        /* position within current record*/
      2 rel_recno fixed bin (31),    /* ordinal record no. in file      */
      2 rwlock bit(8),               /* Read/write concurrency lock     */
      2 access like access_bits,     /* Accesses allowed on file        */
      2 pos_in_parent fixed bin,     /* position in parent              */
      2 parent_bra fixed bin (31),   /* BRA of parent directory         */
      2 hash_thread fixed bin,       /* hash thread                     */
      2 quota_blk_ptr fixed bin,     /* Quota block pointer             */
      2 dir_blk_ptr fixed bin,       /* Directory block pointer         */
      2 dam_idx_ra fixed bin (31),   /* current r.a. in DAM index       */
      2 spare fixed bin;


dcl 1 status_bits based,        /* VSTAT definition                          */
      2 modified bit (1),       /* modified                                  */
      2 sysuse bit (1),         /* open for system use                       */
      2 shtbit bit (1),         /* device shut down                          */
      2 no_close bit (1),       /* special file, not closed by C -ALL */
      2 disk error bit (1),     /* disk error occurred                       */
      2 file_type bit (3),      /* Defined below                             */
      2 open_mode bit (8);      /* Accesses which file is opened with */

      file_type:
        sam_ftype     by 0,     /* File types: SAM file   */
        dam_ftype     by 1,     /* DAM file               */
        samseg_ftype  by 2,     /* SAM segment directory */
        damseg_ftype  by 3,     /* DAM segment directory */
        dir_ftype     by 4,     /* Directory              */
        acl_dir_ftype by 5,     /* ACL directory          */
        acc_cat_ftype by 6;     /* Access category        */
```

AN ATTACH POINT UTE

```
dcl 1 dir_utcme based,              /* attach point Unit Table Entry */
      2 vstat like status_bits,     /* See definition below          */
      2 bra fixed bin(31),          /* BRA                           */
      2 cur_ra fixed bin(31),       /* current r.a. in file          */
      2 ldevno fixed bin,           /* Logical device number         */
      2 rel_wordno fixed bin,       /* position within current record*/
      2 rel_recno fixed bin(31),    /* ordinal record no. in file    */
      2 access,                     /* Access rights                 */
        3 ring1 like access_bits,   /*    in ring 1                  */
        3 ring3 like access_bits,   /*    and ring 3                 */
      2 pos_in_parent fixed bin,    /* position in parent            */
      2 parent_bra fixed bin (31),  /* BRA of parent directory       */
      2 hash_thread fixed bin,      /* hash thread                   */
      2 quota_blk_ptr fixed bin,    /* Quota block pointer           */
      2 dir_blk_ptr fixed bin,      /* Quota directory block pointer */
      2 acl_bra fixed bin (31),   /* BRA of directory containing ACL */
      2 acl_pos fixed bin,        /* Position of default acl in dir  */
      2 spare fixed bin;
```

FLOW OF CONTROL IN THE FILE SYSTEM

Following this page is pseudo-code illustrating the sequence of calls made to file system routines to create and write data to a file.

- o CALL SRCH$$ to create (and open) the file.
- o CALL PRWF$$ to write data to the newly created file.

OVERVIEW OF FILE SYSTEM ROUTINES

Before covering the specifics of the file system routines called to create a file and write data to that file, a general description of each of the routines is presented below:

SRCH$$ -- opens, closes, deletes, and checks the existence of files

FIL-OP -- opens a file and sets up the UTE after the initial record(s) for the file are allocated and the directory entry is created on disk

SGD$OP -- opens a segment directory subfile

ADD-ENT -- adds a new entry to a directory once the initial record(s) for the file are allocated

ALC-REC -- allocates initial record(s) for a new file (or directory) and adjusts record pointers, as necessary

GETREC -- gets a free record in a logical partition by searching the DSKRAT

PRWF$$ -- moves data to and from files as well as performing file positioning

LOCATE -- keeps copies of disk records in memory in order to minimize disk operations

CREATING A FILE

```
SRCH$$:
  Call FIL_OP to create the file

    FIL-OP:
      If (caller supplied unit number)
        Then do
           If (unit number invalid)
             Then return (E$BUNT)
           If (unit in use)
             Then return (E$UIUS)
        End
      Take FSLOK for reading
      Take UFDLOK for writing
      Call ADD_ENT to create the file (entry)

        ADD-ENT:
          If (user does not have add rights)
            Then return (E$NRIT)
          Call ALC_REC to allocate disk record(s)

            ALC-REC:
              Call GETREC to get a disk record

                GETREC:
                  Take RATLOK for writing
                  Hint = RAT word containing bit for UFD record
                  If (RAT bit representing hint >= RAT bit
                      representing the first available record on that
                      partition)
                    Then
                      If (free bit in RAT word holding hint bit)
                        Then do
                          Calculate RA
                          Call LOCATE to write modified RAT record
                          Release RATLOK
                          Return (RA)
                        End
                        Else
                          If (free bit in RAT record holding hint bit)
                            Then do
                              Calculate RA
                              Call LOCATE to write modified RAT record
                              Release RATLOK
                              Return (RA)
                            End
```

CREATING A FILE (CONT'D)

```
                GETREC (cont'd):
                  If (an available record somewhere in that partition)
                    Then do
                      Calculate RA
                      Call LOCATE to write modified RAT record
                      Calculate new first available record in partition
                      Release RATLOK
                      Return (RA)
                    End
                  Release RATLOK
                  Return (E$DISK_FULL)

            ALC-REC (cont'd):
              Call LOCATE to acquire buffer for new record
              Initialize the record header in the BCB
              If (DAM or SEGDAM)
                Then do
                  Call GETREC to get the first data record
                  Call LOCATE to get the index record
                  Set DAM index to point to new data record
                  Call LOCATE to acquire buffer for new data record
                  Initialize the record header in BCB
                End
              Return (new RA)

         ADD-ENT (cont'd):
           Build memory image of file entry
           Write new file entry to UFD record on disk
           Update DTM of parent
           Return (BRA)

    FIL-OP (cont'd):
      Set RWLOCK
      Build memory image of UTE
      If (DAM or SEGDAM)
        Then (set first data record address as UTE.CUR_RA and lowest
              level index record address as UTE.DAM_IDX_RA)
      Allocate a UTE
      Copy UTE image to UTE
      Release UFDLOK and FSLOK
      Return (unit)

SRCH$$ (cont'd):
  If (user did not supplied unit number)
    Then (return (unit))
  Return
```

CREATING A SEGMENT DIRECTORY SUBFILE

```
SRCH$$:
  Call SGD$OP to create segment directory subfile

    SGD$OP:
      If (caller supplied unit number)
        Then
           If (not a valid unit number)
             Then (return (E$BUNT))
           If (unit is in use)
             Then (return (E$UIUS))
      Take FSLOK for reading
      Take UFDLOK for reading
      Take a SDLOK for writing
      Call ALC_REC to allocate a disk record

        ALC-REC:
          Call GETREC to get a disk record

            GETREC:
               .
               .
               .

        ALC-REC (cont'd):
          .
          .
          .
          return (BRA)

  SGD$OP (cont'd):
    Call SGD_WE to write the BRA into segment directory
    Build the UTE image in memory
    If (DAM subfile)
      Then (set first data record address as UTE.CUR_RA and lowest
            level index record address as UTE.DAM_IDX_RA)
    Allocate a UTE
    Copy UTE image to UTE
    Release SDLOK, UFDLOK and FSLOK
    Return (unit)

SRCH$$ (cont'd):
  If (user did not supplied unit number)
    Then (return (unit))
  Return
```

WRITING DATA TO AN EMPTY FILE - PRWF$$

```
PRWF$$:
  Take FSLOK for reading
  If (file not open)
    Then (return (E$UNOP))
  Take a TRNLK for writing
  Pick up the number of words of data to be written
  Set the LOCATE key to RCD_MODIFIED
  Position file to appropriate record
  Call LOCATE to read record into LOCATE buffer

    Do While (there is data to write)
      If (enough room in data record for all the user's data)
        Then do
          Move the data from user's buffer to the LOCATE buffer
          Update UTE.REL_WORDNO
        End
        Else do
          Move as much data as will fit into the LOCATE buffer
          Call ADD_REC to extend the file
          Call LOCATE to acquire buffer for new record
          Update UTE.CUR_RA and UTE.REL_WORDNO
          Update number of words of data left to write
        End
    End

  Call LOCATE to 'forget' the LOCATE buffer
  Release all locks
  Return
```

CLOSING AND DELETING A FILE

Since many of the operations involved in closing and deleting a file simply reverse opening and creating a file, only a list of the routines is presented.

CLOSING

SRCH$$ calls either CLO$FN or CLO$FU:

- o CLO$FN closes a file by name by calling CLOSE.

- o CLO$FU closes a file by file unit by calling CLOSE.

- o CLOSE closes either by name (ldev/BRA) or by unit number and, in both cases, nullifies the UTE pointer in the user's unit table.

DELETING

SRCH$$ calls FIL$DL to delete a file or a directory or SGD$DL to delete a segment directory subfile:

- o FIL$DL attaches to the named object's parent and searches for the entry in the current directory. If the entry is found and the user has delete rights, then the entry is removed from the directory and all records associated with the entry are released. Supporting routines called by FIL$DL are:

  - o ENTINDIR to attach to parent,
  - o FIND_ENT to find the entry in the parent directory,
  - o DEL_ENT to delete the directory entry, and write out a vacant entry.
  - o FREE_REC to release each disk record, starting at the BRA, and calling RTNREC to adjust the DSKRAT for each freed record.

- o SGD$DL reads the BRA of the entry, deletes the entry by clearing the BRA, and then releases all records associated with the subfile. Supporting routines called by SGD$DL are:

  - o SGD_RE to read in the subfile's BRA,
  - o SGD_WR to write out the modified record containing the cleared BRA of the subfile being deleted,
  - o FREE_REC to release the disk records.

Appendix A - Primos Segment Usage

PRIMOS SEGMENTS - DTAR0

0	I/O map segment	[KS>SEG0.PMA]
1	I/O map segment	
2	movutu	
3	movutu	
4	PIC, PCBs, fault handlers, checks, SEMCOM, vpsd	[KS>SEG4.PMA]
5	ring 0 gate segment	
6	ring 0 kernel code and linkage	
7	TFLIOB buffers	(TFLSN1)
10	third segment for kernal code and linkage	
11	file system code and linkage	(LCSEG$)
12	network system code and linkage	(NETSG$)
13	command loop segment 1	
14	PAGCOM, HDRBUF, config, RSAV, FIGCOM, MMAP, tape-dump, warm/cold start code	[SEG14.PMA]
15	second segment for kernal code and linkage	
16	comms code and linkage	
21	DMQ buffers	(DMQBUF)
22	General Event Monitor buffers	
23	SMLC map segment	
24	SMLC map segment	
25	SMLC map segment	
26	SMLC map segment	
27	network buffers	(NETBF$)
30	network queues	(NETBH$)
31	network, SNA code	
32	command loop segment 2	
33	MMAP	
34	named semaphores data area	
35	logout notification queues, CPS	
36	second TFLIOB buffers	(TFLSN2)
37	ACL data area	
41	Command loop segment 1	
50	associative buffers	(BUFSEG)
51	associative buffers	
52	associative buffers	
53	associative buffers	
54	SNA (interactive) data bases	
60	TFLIOB buffer segment #3	
61	TFLIOB buffer segment #4	
62	TFLIOB buffer segment #5	
63	TFLIOB buffer segment #6	
67	RJE code and linkage	
70	RJE code and linkage	
71 . 100	RJE buffers	

PRIMOS SEGMENTS - DTAR0 (continued)

101		
.	32 network mapped segments	
140		
141	DPTX code and linkage	
142	additional DPTX code and linkage	
143		(DPTCOM)
.	DPTX buffers	
200		
201		(PUDCM$)
.	mapped per-process ring 0 stacks	
577		
600		
.	HMAPs/LMAPs or PMTs	
617		
620		
.	dynamically allocated by GETSN$	
717		

PRIMOS SEGMENTS - DTAR1

2000
.
. shared code
.
2577
2600
. dynamically allocated by GETSN$
.
2677

PRIMOS SEGMENTS - DTAR2

4000
. user procedure and linkage, dynamic memory
4777

PRIMOS SEGMENTS - DTAR3

6000	user profile stuff, UPCOM, page fault (wired ring 0) stack, SDTs for DTARS 2 and 3, mapped LOCATE buffer ('17600)	
6001	abbrevs, shared library linkage	
6002	CLDATA, ring 3 stack	(PUSTAK)
6003	unwired ring 0 stack	
6004	CPL work area	
6005	global variables	
6006	additional shared library linkage	
6007		(DYSNBG)
.	dynamically allocated by GETSN$	
.		
6011	ROAM work area	
6012		
.	dynamically allocated by GETSN$	
6014		

Appendix B - Lab Exercises

EXERCISE 1

Directions: Answer the following questions using VPSD, source code, the Ring0 or Ring3 load maps, and what you have learned about Primos.

1) What is the name of the variable whose value indicates the maximum number of virtual segments available for the entire system? Locate this variable's value in memory.

2) How many DTAR0 segments are enabled for this revision of Primos? (HINT: locate DTAR in the map).

3) How many DTAR1 segments are enabled for this revision of Primos?

4) To which segments from 0 to 50 in DTAR0 do you as a ring3 user have access rights? If you do, what are the access rights? (HINT: Locate SDW0 in the map - this is the live SDT for DTAR0).

5) What is done to the STLB before a page-out? Why?

EXERCISE 2

Directions: Answer the following questions using VPSD, source code, the Ring0 or Ring3 load maps, and what you have learned about Primos.

1) How many DTAR2 segments are enabled for your process at this revision of Primos? Can you access all those segments?

2) How many DTAR3 segments are enabled at this revision of Primos?

3) Locate and dump the Ready List in memory.

   a) Who is on the Ready List?

   b) Dump your level on the Ready List until you see your PCB. How many processes are also on your level?

4) Chap your process down a level by changing the priority level in your PCB. What happens, and why?

5) Ask the instructor to spawn the CPL program EXERCISE.2.5.CPL from PI>CLASS as a phantom. Note the user number.

   a) Locate the HOLD state semaphores in memory.

   b) Monitor the queues and watch to see if the phantom process appears on any of the queues.

   c) Based on what you saw or didn´t see, what can you conclude about your phantom process?

6) Spawn the program, PI>CLASS>EXERCISE.2.5.CPL, as a phantom from your process. Access the phantom´s PCB abort flags and change the value to <u>4</u>. What happens?

7) Locate MAXSCH in memory. What is its value?

8) Locate your PCB in memory. Access the abort flags and change the value to <u>10</u>. What happens?

EXERCISE 3

Directions: Answer the following questions using VPSD, source code, the Ring0 or Ring3 load maps, and what you have learned about Primos.

1) Locate you process´ IRB and ORB in memory.

2) Are any processes currently waiting for queue request blocks?

3) You are having a problem with loss of terminal data on input and are unsure as to whether the problem is with your IRB or the tumble tables. There is a counter that keeps track of the number of times the tumble tables have overflowed since coldstart. Normally, the counter is zero (i.e., no tumble table overflow). If the counter is zero, then the problem is the IRB. If the counter is non-zero, then you have a problem with the tumble tables, and possibly, with your IRB as well. But, if the tumble table problem was eliminated and the problem persisted, then it´s probably the IRB. QUESTION: What is the name of this counter?

EXERCISE 4

Directions: Answer the following questions using VPSD, source code, the Ring0 or Ring3 load maps, and what you have learned about Primos.

1) Copy the program EXERCISE.4.1.FTN from PI>CLASS to your directory. EXERCISE.4.1.FTN does a call to TNOU to print out 'HELLO' at the terminal and then calls EXIT. You are going to verify that the link to TNOU is dynamically snapped at runtime. Compile EXERCISE.4.1.FTN with -64V and -EXPLIST. EXPLIST will generate an expanded listing of the FTN statements and the PMA instructions generated by each one. Do a normal load, but be sure to get a map. Spool off both EXERCISE.4.1.LIST and the runfile map. Then invoke the runfile by typing SEG EXERCISE.4.1 1/1. This causes VPSD to be loaded with your runfile, as well.

   a) What is the offset in EXERCISE.4.1'S link base to which the 2-word PCL instruction, generated by the call to TNOU, is pointing.

   b) What is the contents of the LB location you found in (a)? This will be a 2-word address so make sure you get both the segment number and the word offset.

   c) Go to the address you found in (b) and display its contents and the contents of the next couple of locations. What are you looking at?

   d) Set a breakpoint on the PCL instruction for the CALL EXIT statement at location 1006 in the PB. Then execute the program.

   e) You will see HELLO and on the next line, an indication that the breakpoint at 1006 has been reached (i.e., you have executed the PCL for TNOU and are 'waiting' on the PCL instruction for

      the call to EXIT. Now, go back into the link base and access the same location you accessed in (b). What is the address you see now? What has happened?

   f) Continue execution of EXERCISE.4.1.

EXERCISE 4 IS CONTINUED ON THE NEXT PAGE.

EXERCISE 4 (continued)

2) Copy the file EXERCISE.4.2.CPL from PI>CLASS to your directory. EXERCISE.4.2.CPL compiles and loads EXERCISE.4.2.F77. When EXERCISE.4.2.CPL terminates execution, issue a RLS -ALL command to make sure you clean up your ring3 stack. Then execute EXERCISE.4.2.SEG. Open up a como file and issue the DMSTK command, specifying -ALL and -ON_UNITS as arguments. Then close and spool your como file.

   a) Using the RING3 and RING0 maps, determine which routines are represented by the stack frames in the DMSTK output in your como file.

   b) Based on (a), what sequence of events occurred?

   c) To check your answer to (b), copy EXERCISE.4.2.F77 and FAKE.PMA from PI>CLASS to your directory. Examine.

EXERCISE 5

1. Execute the RLS -ALL command. Then execute the LD command on a directory of your choice.

   a) Locate the starting address of CLDATA in the ring 3 map.

   b) Copy CLDATA.INS.PMA to your own directory. Remove the NLST pseudo-op (about the fifth line), save and assemble. Examine CLDATA.INS.LIST and locate the offset from the beginning of CLDATA to SMTLPT(2). Add that offset to the address of CLDATA found in (a) above. This is the address of your process´ first SMT block.

   c) Find the SMT for LD. (Hint: look at the pathname field).

   d) How many DTAR2 segments are used for LD´s procedure code and linkage?

   e) Where in DTAR2 is LD´s procedure and linkage?

   f) Verify your answers by executing LE LD.RUN -DET.

Extra

Execute LD on a large directory such as PRIMOS>KS. Hit ^P. Execute the LD command again.

   a) Determine the DTAR2 segment(s) used for the second invocation´s linkage.

   b) Locate the first invocation´s saved linkage segment number (Hint: look at the layout of SMT_ACTIVE_ENT in PRIMOS>INSERT>EPFFMT.INS.PLP).

Appendix C - Miscellaneous

READING THE SYSTEM LOAD MAPS

```
                 seg     seg
                 num    offset
                  |       |
                  V       V
PPNLST          0014    000567   OTHER
NSEG            0014    000614            COMMON
PAGCOM          0014    000614            COMMON
NUSEG           0014    000615            COMMON
PFSW            0014    000616   OTHER
USRLEV          0014    000625   OTHER
```

```
              A(ECB)             STARTING          NUM OF     NUM OF        LB
                 |                  PB            WORDS OF   WORDS OF    SETTING
                 |                  |              STACK     LINKAGE       |
                 |                  |                |          |          |
                 V                  V                V          V          V
PA$SET      0011   123732     0011   120732       002534     000212    0011   123312
PABORT      0006   040744     0006   037672       000040     000222    0006   040322
PAG$FS      0006   033403     0006   033242       000102     000041    0006   033000
```

VPSD COMMAND SUMMARY

SN segment-number -- sets segment number

A [:format-symbol] [value] [:new-format-symbol] terminator -- accesses a location relative to current segment

format-symbol	
:A	ASCII
:B	Binary
:D	Decimal
:H	Hexadecimal
:O	Octal
:S	Symbolic

value	
n	absolute
*	current
*+n	relative
*-n	relative

terminator	
CR	*+1
,	*+1
^	*-1
.nCR	*+n
.-nCR	*-n
/	return, remember *
?	return, remember *
!	return, forget *

Q -- quit from VPSD and return to command level

D start-offset ending-offset [:new format symbol] -- dump a block of locations relative to current segment

B offset -- set a breakpoint at specified offset relative to current segment

EX -- execute a runfile from the start

PR -- proceed with execution from current breakpoint

VPSD DEMONSTRATION

```
OK, FTN HELLO -64V -EXPLIST
0000 ERRORS [<.MAIN.>FTN-REV19.2.2]

OK, SEG -LOAD
[SEG rev 19.2.2]
$ LO HELLO
$ LI
LOAD COMPLETE
$ SA
$ MA HELLO.MAP
$ Q

OK, SEG HELLO 1/1                          /* '1/1' LOADS IN VPSD
                                           /* '$' IS VPSD'S PROMPT
$SN 4001 C/R                               /* SN = SET THE SEGMENT NUMBER TO 4001

$A 1000 C/R                                /* A = ACCESS LOCATION 4001/1000
4001/ 1000 PCL% LB%+ 422,* C/R             /* THE DEFAULT DISPLAY MODE IS SYMBOLIC
4001/ 1002 AP    1010,S C/R                /* TO DISPLAY THE NEXT LOCATION, SIMPLY
4001/ 1004 AP   LB%+ 400,SL C/R            /* TYPE A CARRIAGE RETURN.VPSD DOES NOT
4001/ 1006 PCL% LB%+ 424,* C/R             /* UNDERSTAND THE 'ERASE' CHARACTER AND
4001/ 1010 LDA#  305,*X C/R                /* WILL GIVE YOU AN ERROR ('E') AND
4001/ 1011 ANA#  314,*X C/R                /* THE PROMPT. YOU MUST RETYPE THE LINE
.
4001/ 1012 ANA#  653,*X C/R
4001/ 1013 JST#  240,*    ^                /* TO ACCESS THE PREVIOUS LOCATION,TYPE
4001/ 1012 ANA#  653,*X   ^                /* THE CAROT (^) CHARACTER INSTEAD OF A
4001/ 1011 ANA#  314,*X   ^                /* CARRIAGE RETURN.
4001/ 1010 LDA#  305,*X   ^
4001/ 1007 DAC   424 ^
4001/ 1006 PCL% LB%+ 424,*    ^
4001/ 1005 AP   SB%+ 61432 ^
4001/ 1004 AP   LB%+ 400,SL ^
4001/ 1003 E32I  ^
4001/ 1002 DAC   100 ^
4001/ 1001 DAC   422 ^                        /* TO CHANGE THE DISPLAY MODE FROM
4001/ 1000 PCL% LB%+ 422,*        :O C/*    SYMBOLIC TO OCTAL, TYPE ':O'
4001/ 1002 100 :D C/R                         /* DECIMAL REPRESENTATION, TYPE ':D'
4001/ 1003  00520 :H C/R                      /* HEX REPRESENTATION, TYPE ':H'
4001/ 1004 02C0  :S C/R                       /* TO RETURN TO SYMBOLIC, TYPE ':S'
4001/ 1005 DAC   400 /                        /* TO RETURN TO THE '$' PROMPT, TYPE
                                              /* '/' (WITH NO CARRIAGE RETURN).
```

VPSD DEMONSTRATION (continued)

```
                    /* TO DUMP A SERIES OF LOCATIONS, ISSUE
                    /* THE 'D' DIRECTIVE AND SPECIFY BOTH
                    /* THE STARTING AND ENDING LOCATIONS.
                    /* YOU CAN ALSO SPECIFY THE DISPLAY MODE.
                    /* 8 LOCATIONS PER LINE IS DISPLAYED.

$D 1000 1010 :0 C/R
  4001/ 1000  61432    422    100   1010   1300    400  61432    424
  4001/ 1010 144305

$SN 4002 C/R                  /* SWITCH FROM SEGMENT 4001 TO 4002

$A 0 C/R                      /* ACCESS LOCATION 4002/0
4002/ 0 5 C/R
4002/ 1 0   /                 /* TO CHANGE THE CONTENTS OF A LOCATION
$A 1 C/R                      /* SIMPLY ACCESS THE LOCATION. WHEN THE
4002/ 1 0   1 C/R             /* LOCATION IS DISPLAYED, TYPE IN THE NEW
4002/ 2 4001 ^                /* VALUE. IN THE EXAMPLE, LOCATION
4002/ 1 1                     /* 4002/1 WAS ACCESSED. ITS ORIGINAL
4002/ 2 4001 /                /* VALUE WAS 0. IT WAS CHANGED TO BE A 1.

$SN 4001 C/R                  /* SWITCH BACK TO SEGMENT 4001

$B 1006 C/R                   /* SET A BREAKPOINT AT LOCATION
                              /* 4001/1006.  THE PURPOSE OF A
                              /* BREAKPOINT IS TO HALT PROGRAM
                              /* EXECUTION AT A PARTICULAR LOCATION
                              /* SO THAT MEMORY CAN BE EXAMINED.

$EX C/R                       /* TO START PROGRAM EXECUTION,TYPE 'EX'

HELLO                         /* 'HELLO' IS PRINTED OUT BY THE
                              /* PROGRAM.  THE NEXT LINE TELLS US
                              /* THAT EXECUTION IS HALTED AT THE
                              /* BREAKPOINT WE SET ABOVE.
4001/ 1006: PCL% LB%+ 424,*     A=100000 B=212 X=0 K=14100 R=0 Y=26430

$PR C/R                       /* TO CONTINUE EXECUTION, TYPE 'PR'
OK,
```

Appendix D - Acronyms

ACRONYMS

ACRONYM	MEANING	SECTION COVERED
ALU	Arithmetic Logic Unit	Hardware
AMLC	Asynchronous Multi-Line Controller	Device
ARGT	Argument Transfer	Procedure
AST	Assigned Segment Table	EPFs
BMA	Bus Memory Address	Hardware
BMC	Bus Memory Control	Hardware
BMD	Bus Memory Data	Hardware
BPA	Bus Peripheral Address	Hardware
BPC	Bus Peripheral Control	Hardware
BPD	Bus Peripheral Data	Hardware
BRA	Beginning Record Address	File System
CALF	Call Fault Handler	Exceptions
CF	Condition Frame	Exceptions
CIB	Critical Information Block	EPFs
CTI	Character Time Interrupt	Device
DFU	Dynamic File Units	File System
DMA	Direct Memory Access	Device
DMC	Direct Memory Channel	Device
DMQ	Direct Memory Queue	Device
DMT	Direct Memory Transfer	Device
DP	Diagnostic Processor	Hardware
DSKRAT	Disk Record Availability Table	File System
DTAR	Descriptor Table Address Register	Memory
DTB	Data Template Block	EPFs
EPF	Executable program Format	EPFs
FADDR	Fault Address	Exceptions
FF	Fault Frame	Exceptions
FCODE	Fault Code	Exceptions
FIM	Fault Intercept Module	Exceptions
HMAP	Hardware Map	Memory
ICS	Intelligent Controller Subsystem	Device
IOTLB	I/O Table Lookaside Buffer	Memory
IRB	Input Ring Buffer	Device
LB	Linkage Base	Procedure
LDEV	Logical Device Number	File System
LMAP	Logical Map	Memory
LTD	Linkage Template Descriptor	EPFs
LTE	Linkage Template Entry	EPFs
MMAP	Memory Map	Memory
MPC	Micro Programmable Controller	Hardware
ODB	On-Unit Descriptor Block	Exceptions
ORB	Output Ring Buffer	Device
PB	Procedure Base	Procedure

ACRONYMS (cont'd)

ACRONYM	MEANING	SECTION COVERED
PCB	Process Control Block	Process
P-ctr	Program Counter	Hardware
PCL	Procedure Call	Procedure
PIC	Phantom Interrupt Code	Device
	Programmable Interval Clock	Device
PIO	Programmed Input/Output	Device
PMT	Page Map Table	Memory
PPA	Pointer to Process A	Process
PPB	Pointer to Process B	Process
PPN	Physical Page Number	Memory
PRTN	Procedure Return	
QRB	(Disk) Queue Request Block	Device
RA	Record Address	File System
RF	Register File	Hardware
ROIPQNM	RO Input Queue Notification Mechanism	Device
SB	Stack Base	Procedure
SDT	Segment Descriptor Table	Memory
SDW	Segment Descriptor Word	Memory
SMT	Segment Mapping Table	EPFs
SOC	System Option Controller	Hardware
STLB	Segment Table Lookaside Buffer	Memory
SWI	Software Interrupt	Exceptions
UART	Universal Asynchronous Receive Transmit	Device
U-CODE	Microcode (firmware)	Hardware
URC	Unit Record Controller	Hardware
UTE	Unit Table Entry	File System
VCIB	Very Critical Information Block	EPFs
VCP	Virtual Control Panel	Hardware
VMFA	Virtual Memory File Access	Memory

Appendix E - Reading List

READING LIST

SG-194	Primos Student Guide
DOC9473-1PA	System Architecture Reference Guide
DOC3621-190P	Subroutines Reference Guide
DOC6904-191P	Prime 50 Series Technical Summary

Hardware Features	DOC9473-1PA	1
	DOC6904-191P	2; 12:1-7
Memory Management	DOC9473-1PA	2:1-7; 3:1-34; 4:1-25
	DOC6904-191P	4:1-15
	SG-194	2
Process Management	DOC9473-1PA	9:1-30
	DOC6904-191P	3:1-10
	SG-194	3
Device Management	DOC9473-1PA	10:3-5; 11:1-17
	DOC6904-191P	5; 10:4-5
	SG-194	4
Procedure Management	DOC9473-1PA	8:1-15
	DOC6904-191P	8:1-8
	SG-194	5
Exception Handling	DOC9473-1PA	10:6-16
	DOC6904-191P	8:7-10; 10:4
	DOC3621-190P	22:1-6,(7-15), 16-24,25-43),43-53
	SG-194	6
Command Environment	DOC6904-191P	9:4-10, (11-12)
	SG-194	7
File System	DOC3621-190P	I:1-24
	DOC6904-191P	6:1-9
	SG-194	9