PROPERTY OF DATA GENERAL CORPORATION

CSE

AOS Internals

Reference Manual

(AOS Revision 3.11)

PROPERTY OF DATA GENERAL CORPORATION

NOTICE

Revision 3.02 - November 1979 (Original release)
Revision 3.03a - January 1980
Revision 3.03b - March 1980
Revision 3.11 - June 1980

In nova fert animus
Mutatas dicere causas

OVID

TABLE OF CONTENTS

CHAPTER 3 - DATABASES, DATA STRUCTURES, DATA RESOURCES

CHAPTER 4 - PATHS AND STATES

CHAPTER 5 - DISKWORLD

CHAPTER 6 - SYSTEM CALL PROCESSING

CHAPTER 7 - INTERPROCESS COMMUNICATIONS AND CONNECTIONS

CHAPTER 8 - GHOST

CHAPTER 9 - PMGR, THE PERIPHERALS MANAGER

CHAPTER 10 - EXEC

CHAPTER 11 - CLI

CHAPTER 12 - SYNC WORLD

CHAPTER 13 - MISCELLANEOUS TOPICS

CHAPTER 1 - INTRODUCTION

(upcated for AOS Rev. 3.11)

NOTE: The information in this manual is relevant for rev 3.11 of AOS only and is subject to change in subsequent revisions.

By definition, a modbits system is an M600 with the hardware page modified option enabled.

1.1 Foreword

The following pages describe the new format for the AOS internals manual. Some chapters have been synthesized from various pieces of existing documentation, with the holes filled in by new pieces of our own writing. Some chapters have been completely rewritten. In most cases we got rid of all historical information about previous releases of the software, in order to give an image of the product as it stands now. We also tried to gather all information about a particular subject in one single place. This approach should facilitate the task of the reader, and also help keep the manual up to date for every future update of the software.

These notes are intended to provide the reader with a usable reference document on AOS as well as to give him or her a general understanding of the workings of this operating system and its interactions with the users. They are designed as introductory material and should help in the consultation of the listings.

The listings are the final arbiter anyway. When in doubt, go to them. And do not forget that the code itself prevails over the comments!

1.2 Introduction to operating systems

Functions:

* Resource manager

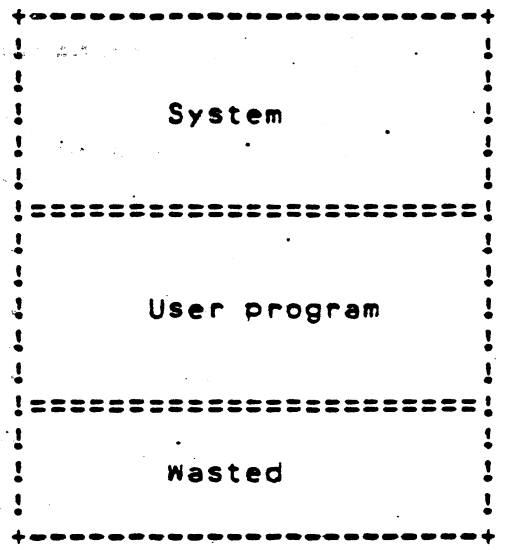
  Memory
  Processors
  Devices
  Information

* Interface to hardware

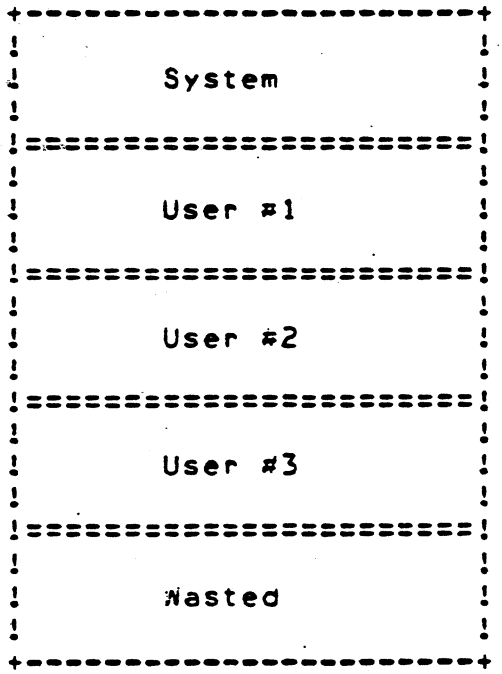
* Protection

Memory Management:

* Simple contiguous allocation



* Partitioned



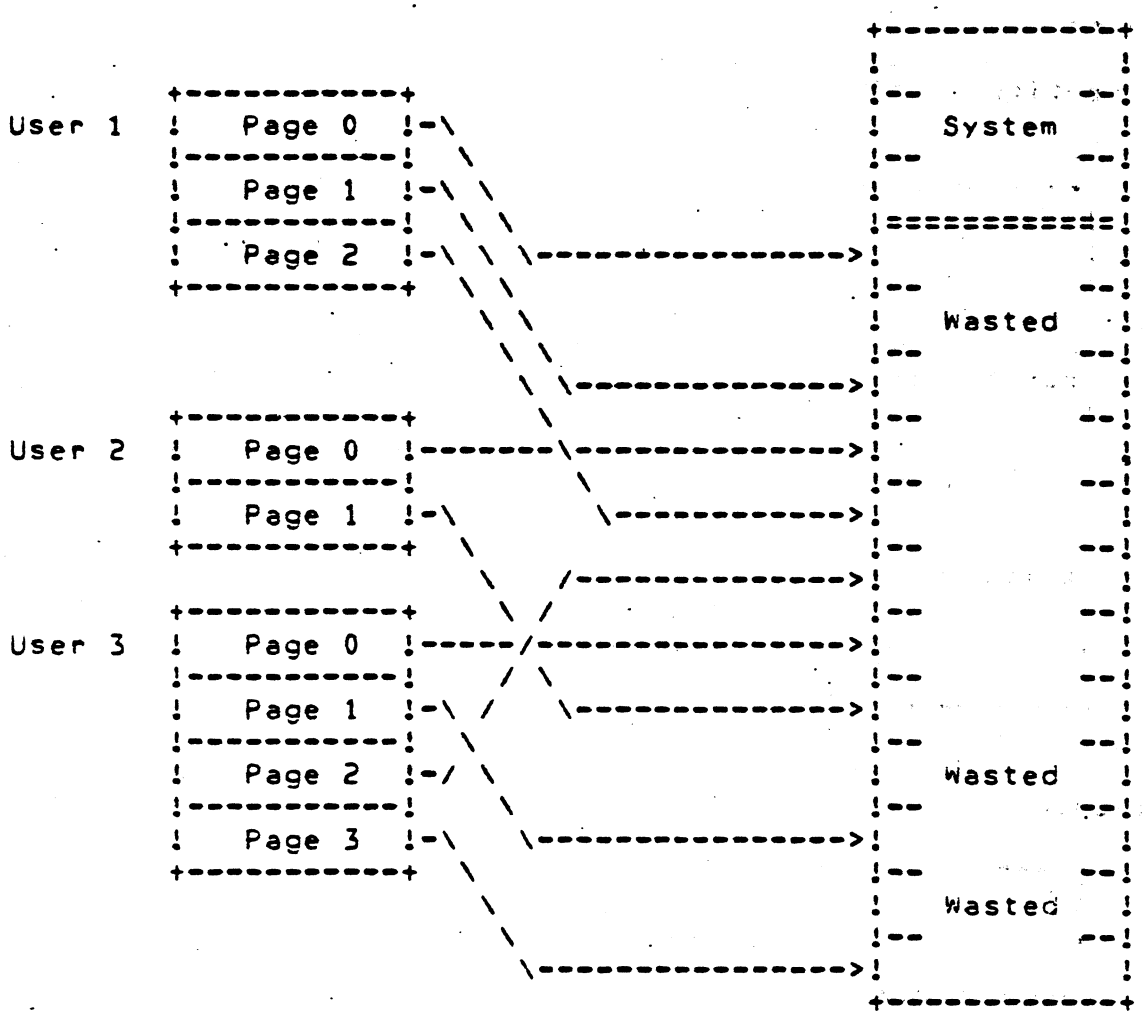
Advantages:

1. less wasted memory
2. less wasted CPU time (multiprogramming)

Disadvantages:

1. Special hardware
2. OS is more complex
3. Fragmentation

* Paged allocation



Advantages:

1. Solves fragmentation problem

Disadvantages:

1. Additional hardware needed (page tables or map registers)
2. Non contiguous programs
3. Entire program must be in memory

* Demand paging

  Advantages:

  1. Allows partially loaded programs to be executed
  2. Additional hardware (beyond paged allocated additions) to provide for referenced and modified flags, fault flag, and the restarting of instructions after a page fault.

  Disadvantages:

  1. Different coding philosophies (i.e. minimal indirection, modular code)
  2. Extra overhead
  3. Thrashing

Processor management techniques

* Run to completion

  1. Simple to implement
  2. Adequate only for stand-alone or batch operating systems
  3. Need ability to terminate run-away programs
  4. CPU time is wasted waiting for completion of I/O.

* Run until blocked (pended waiting for I/O completion)

  1. Allows multiprogramming
  2. CPU bound programs can monopolize the CPU

* Time-slice

  Run until either:

  1. Process blocks
  2. The time slice expires

  Possible algorithms:

  1. First come, first serve
  2. Round robin
  3. Priority

Device Management

* Types of devices:

    1. input
    2. output
    3. storage - serial (tape)
                 direct access (disk)

* Technique

    1. Dedicated allocation
    2. Spooling
    3. Shared access

* Problems

    1. Devices are expensive --> sharing helps in solving this prob.
    2. Device speeds vary but all are slow compared to the processor speeds.
    3. Error handling.

Comparison of Contiguous File allocation versus Indexed

Contiguous:

Advantages:

1. Simple
2. I/O is fast, efficent

Disadvantages:

1. Fragmentation
2. Difficult to grow files
3. Must allocate disk space for 'holes'

Indexed:

Advantages:

1. Solves fragmentation problem
2. Easy to grow files
3. Need not allocate disk space for 'holes'

Disadvantages:

1. Several accesses may be required to get the data.
2. More disk space required for a given size file.

Resource deadlocks

1.3 AOS Building blocks

Section 1.2 discussed the principle functions of an operating system. The most important, is resource management. In AOS, resources are handled by a number of different 'programs'. The following discussion will specifically tie parts of AOS to the resources they handle. This will also serve as an introduction to the parts of AOS.

1. The KERNEL

This is the heart of AOS; it is the code that is created by the AOSGEN program. The KERNEL is responsible for scheduling (process management), file / memory management, and interrupt processing.

2. The PMGR

All character oriented device that are not on the data channel or BMC are controlled by the PMGR (peripheral manager). These include the consoles, non-data channel printers (LPA / LPC), card readers, and paper tape punch/readers. The PMGR exists in three forms, one for the standard AOS configuration, one for those with ASYNC DCUs, and one for M600's with and IOP.

3. EXEC

EXEC is responsible for management of batch / print queues, labelled magtape mounts, and log on / log off.

4. The GHOST

The GHOST, which is actually an extension to the user program is responsible for labelled magtape, system call pre-processing and generic file management.

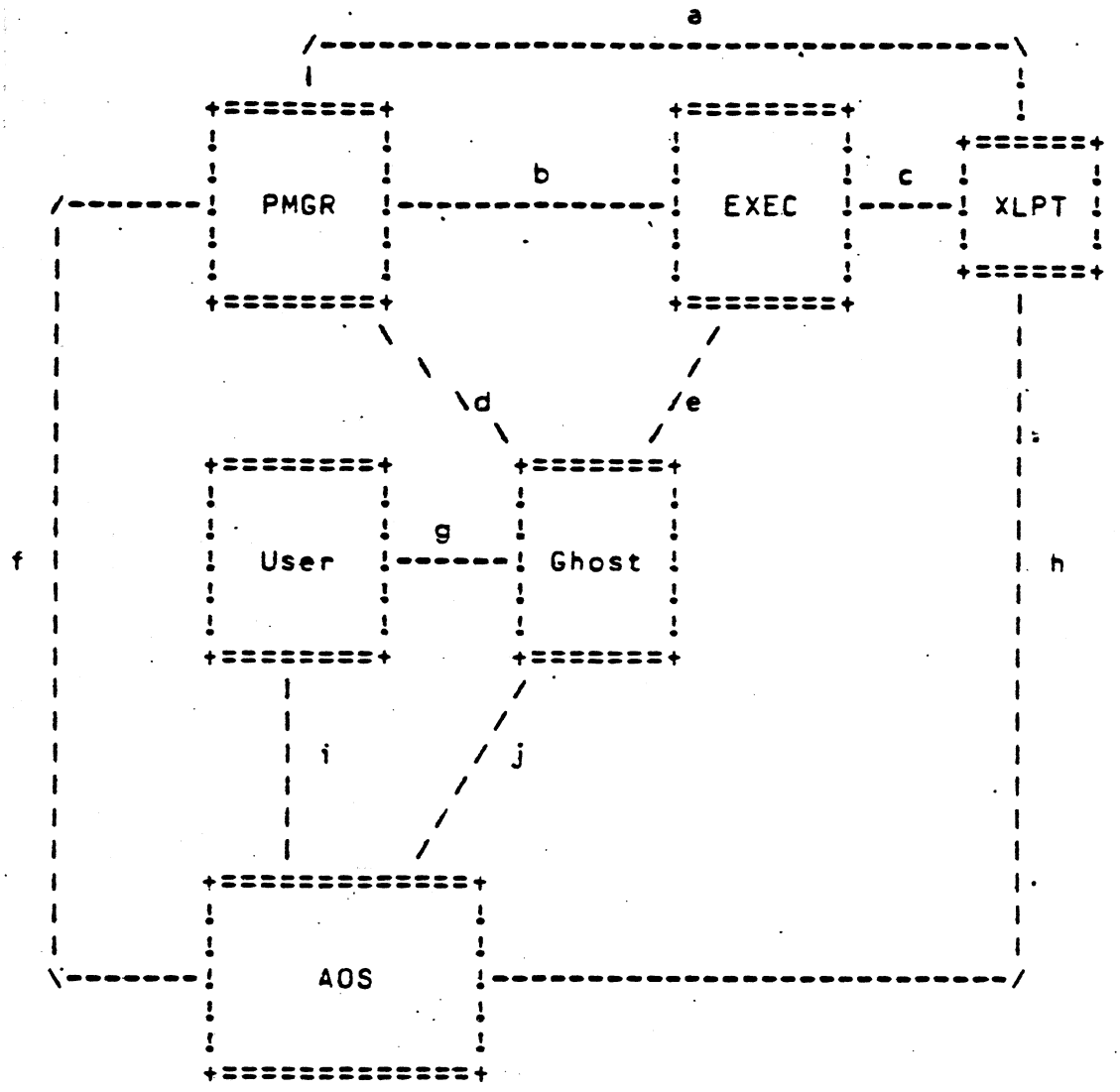
5. The CLI

The CLI is an elaborate system call translator with a large number of bells and whistles (templete expansion for example)

1.4 AOS block diagram

On the following pages, there are two large block diagrams of AOS. The first outlines the overall system picture and does not discuss what goes on inside each module. The second is an attempt to represent the functions performed by the AOS kernel. A complete description of the system would include some "special" users like PMGR, EXEC and the users' ghosts. All these are somehow included in the box called "User events" and will be described at their place, further in this manual.

The Big Picture



a. XLPT <--> PMGR — This interface is used in printing to non-data channel devices.

b. PMGR <--> EXEC — This interface is used for logon/logoff

c. EXEC <--> XLPT — This interface is used for queuing print requests and handling such things as restarts, flushes ...

d. PMGR <--> GHOST — This interface handles I/O to terminals (?READ / ?WRITE are translated into IPC send/rec)

e. EXEC <--> GHOST — This interface handles the ?EXEC system call

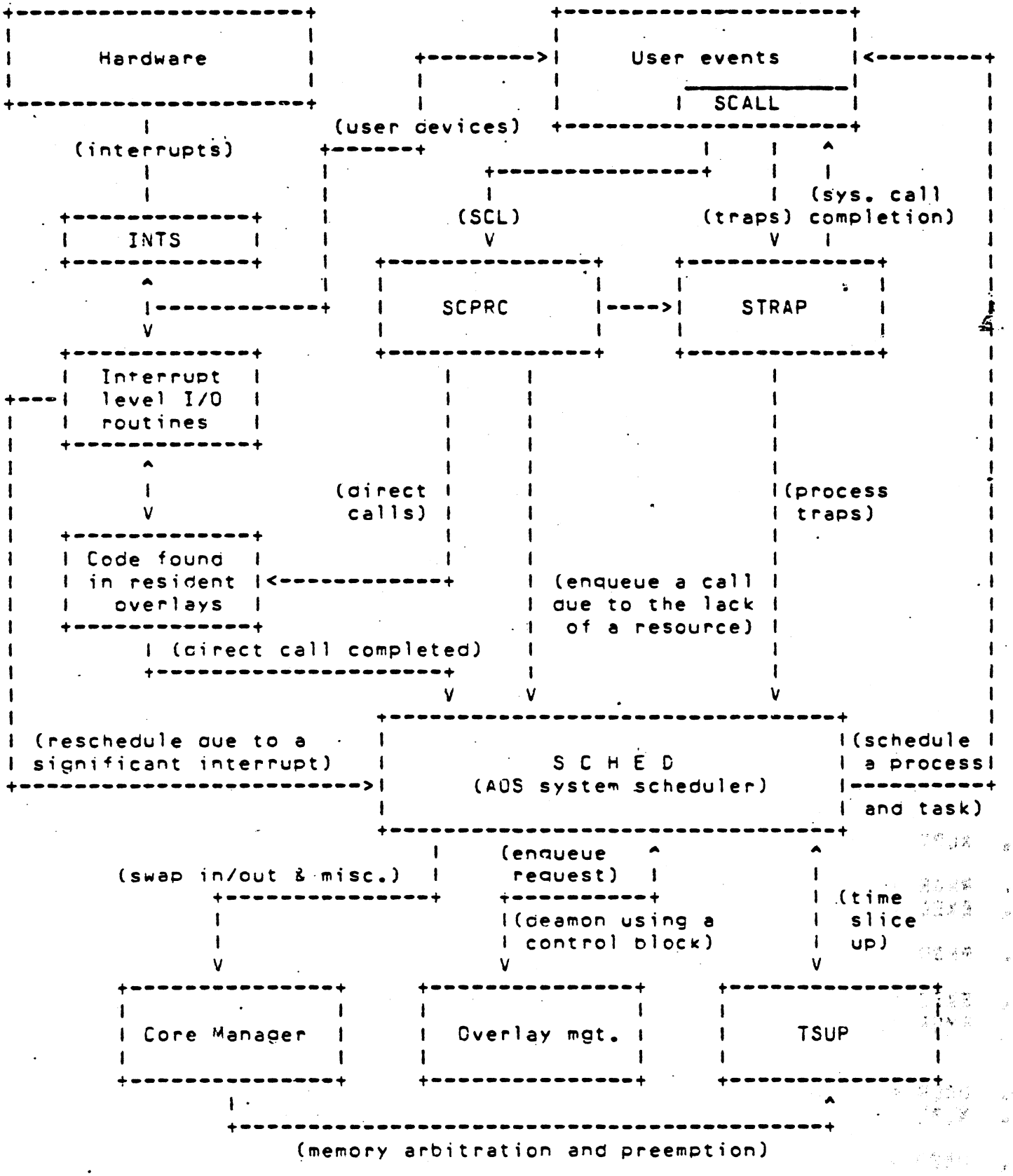
f. PMGR <--> AOS — Character oriented I/O is handled by PMGR at AOS request. In return, the PMGR can request that AOS reschedule users.

g. USER <--> GHOST — GHOST calls are processed through this link.

h. XLPT <--> AOS — This interface is used in printing to data channel devices.

i. USER <--> AOS — Standard system calls (?SYSID, ?GTOD ...)

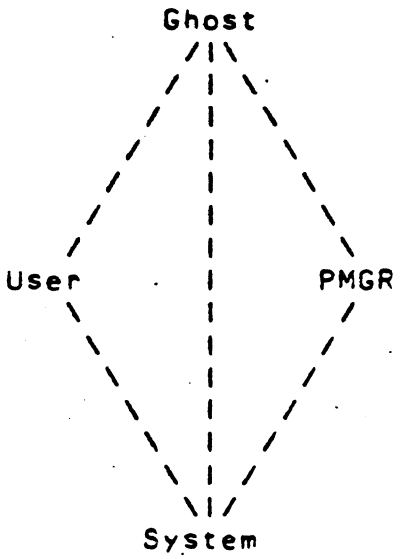
j. AOS <--> GHOST — Many GHOST calls are translated by the GHOST into normal AOS system calls. This interface handles this case


Hardware
User events
SCALL
(user devices)
(interrupts)
INTS
(SCL)
(traps)
(sys. call completion)
SCPRC
STRAP
Interrupt level I/O routines
(direct calls)
(process traps)
Code found in resident overlays
(enqueue a call due to the lack of a resource)
(direct call completed)
(reschedule due to a significant interrupt)
S C H E D
(AOS system scheduler)
(schedule a process and task)
(swap in/out & misc.)
(enqueue request)
(deamon using a control block)
(time slice up)
Core Manager
Overlay mgt.
TSUP
(memory arbitration and preemption)

1.5 User - system interface

Since the ghost and the PMGR exist outside of the AOS address space, the usual system-user interface has several sub-interfaces :

User-system, user-ghost, ghost-system, PMGR-ghost/user, and PMGR-system.



The interfaces involving either ghost or PMGR will be discussed in the ghost and PMGR chapters. The user-system interface is summarized below and its various aspects will be discussed in detail in the appropriate chapters of the manual.

1.5.1 Data interfaces

AOS accesses a user's data in two situations:

- the user has made a system call, in which case there will be data in the AC's and possibly in a packet
- the system needs to look at the user's TCB's.

In the first case, the system receives data from the user's AC's and from his address space by mapping the appropriate slots of the user's context into some pre-determined slots of the system's address space.

In the second case, the system maps page zero of the user's address space into the system's address space.

AOS spontaneously transfers data to a user's space under only one circumstance :

- if a process is aborted, a termination message from the system is sent to the process' father.

1.5.2 Control interfaces

There are three ways AOS can take control away from a user :

- the user makes a system call
- the process traps
- an interrupt from a device comes in.

In the last two cases, control is yanked away from the user by the hardware without any software preparation. The user's AC's and PC are saved.

If a trap occurred, the process will be aborted by the system.

If an interrupt came in, control will be restored to the user after servicing the interrupt, unless the interrupt was due to the user's present allowance of CPU time ("time slice") having ended, in which case someone else will be scheduled. The user will not receive control again until the scheduler gives it to him.

When a user makes a system call, control goes to the SCALL module which has been bound into the user's program. SCALL determines what parameters are required and executes the appropriate version of an SYC instruction. This takes control into the system at location SYST in module SCPRC.

SCALL
SYC
SYST
SCPRC

1.6 Structure description

There are many ways to describe an operating system. Our approach is, from chapter to chapter, to give the reader a view of AOS first from the inside then out to some part of the system, and further out to the users' space.

We shall first describe AOS process image, the system address space and how AOS uses the physical memory of the Eclipse (chapter 2).

We then take a closer look at the insides, describing some important databases, structures of databases, and resources such as overlays (chapter 3).

Looking out from there, we see the world which, for an operating system, consists of users competing for resources and devices requiring service. With algorithms, paths and states (chapter 4) we study this resources contention, the management of processes, the scheduling and the interrupt world.

The diskworld and file system are described separately (chapter 5), as well as the synchronous world (chapter 12).

Next we look at an AOS interface to the user space and discuss the handling of system calls (chapter 6).

The mechanisms of communication between users and with the system are detailed in Interprocess Communications and Connections (chapter 7).

In order to perform some of the things an operating system is supposed to do for a user, AOS implements a number of functions in this part of the user space which we call the Ghost (chapter 8).

Going further out, we find the processes to which AOS detached some more of its O/S duties, which it cannot possibly perform in its own exiguous address space : the PMGR (chapter 9) and the EXEC (chapter 10).

Chapter 11 discusses the CLI Command Language Interpreter.

Chapter 12 discusses the sync world.

Chapter 13 deals with miscellaneous subjects. Currently this includes Networking, AP support, and SYSBOOT.

CHAPTER 2 - CORE IMAGES
(updated for AOS Rev. 3.11)

2.1 Introduction

There are two important ways of looking at an AOS image. One is the picture of all of physical core, the other is the structure and contents of AOS's 32K logical address space.

In this chapter we explain how AOS builds its address space from initialization to dynamic allocation of memory blocks, and list all memory management routines used by the system. The database AOS sets up to keep track of every page of physical memory is described here as well.

We assume the reader is familiar with the concepts of mapping on the Eclipse hardware.

2.2 Initial physical image

After reading the operating system into core but prior to executing any initialization code, physical memory has the .SY file starting at low core, and all remaining pages have nothing in them. This .SY file is composed of AOS .OB's and libraries as illustrated in tables 2.2.1 through 2.2.5. The .SY file varies in size depending on how many devices the system will be driving (the device control tables and device drivers are conditionally loaded from DCTLIB and DEVLIB at AOSGEN BIND time.) If an IOP is present, names enclosed in { } apply.

```
physical loc.          .OB or library

   0           SZERO   - ZREL constants, bridges and pointers
 400           STABLE  - constants, counters, queue headers
2000           SCHED   - scheduler
  !            [DEVLIB.LB] - device drivers
  !            {DEVLIB2.LB}
  !            DUMPR   - core dumper for crashes (resident part)
  !            [LIB2.LB] - resident code
  !            {LIB4.LB}
  !            [DCTLIB.LB] - device control tables
  !            {DCTLIB2.LB}
  !            PANIC   - panic shutdown code
  !            MEMAP   - header for core map entries
  !            SINIT   - system initialization
  !            MLDUI   - master LDU initialization
  !            CLIBT   - CLI bootstrap
  !            IRMG2   - device support tables:DCTTB,UNTTB,PERTB
  !            SCALL   - system call interface on user side
  !            CLIEN   - location marking end of init. code
  V            [LIB3.LB] - system overlays
  ?            {LIB3.LB}
```

Load order of .OB's and libraries

Table 2.2.1

DKBWT	DKBWT
DPEIS	DPEIS
DPAIS	DPAIS
DPASU	DPASU
DPAST	DPAST
DPACT	DPACT
DPDIU	DPDIU
DPDSU	DPDSU
DPDST	DPDST
DPEIU	DPEIU
DPFDR	DPFDR
DPGIU	DPGIU
DPIIU	DPIIU
MCAIO	MCAIO
MTADR	MTADR
DCUDR	
MUXDR	
CDRDR	IUPDR
TTYDR	
LPTDR	
LPBWT	LPBWT
DCUIS	DCUIS
SLMIS	SLMIS
SLMFN	SLMFN
CCTAB	CCTAB
PWRFL	PWRFL

.OB's in DEVLIB (.OB's in DEVLIB2)

Device Drivers in load order

Table 2.2.2

SCPRC	- system call interface on system side	-	SCPRC
STRAP	- trap handler	-	STRAP
SRDB	- shared read block code	-	SRDB
SOVLY	- overlay manager	-	SOVLY
INTS	- interrupt service code	-	INTS
SGSUB	- general use subroutines	-	SGSUB
BHSUB	- buffer header subroutines	-	BHSUB
MAPER	- slot mapping routines	-	MAPER
COREM	- core manager	-	COREM
MEMRY	- GSMEM manager	-	MEMRY
WDBLK	- withdraw disk block	-	WDBLK
WDCBK	- withdraw contiguous disk blocks	-	WDCBK
DEBLK	- deposit disk block	-	DEBLK
DEBKS	- deposit disk blocks	-	DEBKS
CPDCK	- check control point directory size	-	CPDCK
CPDUP	- update control point directory size	-	CPDUP
PERDP	- character peripherals driver		
DSKIO	- I/O routines at the file level	-	DSKIO
REMAP	- bad block remapping routines	-	REMAP
BUFIO	- I/O routines at the buffer level	-	BUFIO
CACHE	- cache buffering routines	-	CACHE
ESD	- emergency shutdown module	-	ESD
STKS	- system stacks	-	STKS

LIB2	- resident code in load order (all write-protected except STKS)	-	(LIB4)

Table 2.2.3

DKBDC,DKB1D,...DKB7D	DKBDC,DKB1D,...DKB7D
DPDDC,DPD1D	DPDDC,DPD1D
DPEDC,DPE1D	DPEDC,DPE1D
DPFDC,DPF1D,...DPF7D	DPFDC,DPF1D,...DPF7D
DPGDC,DPG1D	DPGDC,DPG1D
DPIDC,DPI1D	DPIDC,DPI1D
MTADC,MTA1D	MTADC,MTA1D
MCADC,MCA1D	MCADC,MCA1D
MTBDC,MTB1D	MTBDC,MTB1D
DCUDC	
MUXDC,MUX1D	
CRADC,CRA1D	
PLADC,PLA1D	IOPDC
LPADC,LPA1D	
TRADC,TRA1D	
TPADC,TPA1D	
TTYDC,TTY1D	
SLMDC	SLMDC
SD0DC,...SD3DC	SD0DC,...SD3DC
LPBDC,LPB1D,...LPB7D	LPBDC,LPB1D,...LPB7D
LPCDC,LPC1D	
LPDDC,LPD1D	LPDDC,LPD1D
PWRDC	PWRDC

DCTLIB - Device Control Tables in load order - {DCTLIB2}

Table 2.2.4

0 SSOV1 - ?GTOD,?STOD,?GDAY,?SDAY;init.release code;get/release a UDB
1 UNIT - device dependent routines for disk & magtape(unit open/close)
2 CMOV1 - miscellaneous Core Manager functions
3 SCMOD - direct system calls processing
4 IOOV - user I/O ?RDB,?WRB,?SPAGE
5 SWAPI - swap in extender and shared portion of a process
6 SWAPO - swap out shared portion of a process; preempt a process
7 SWAPM - swap in/out the CCB's & unshared portion of a process
10 ECC - ECC and timeout support for ZEBRA and FHD; enqueue for MHD
11 DVRS - device restart routines after a powerfail (disks & magtape)
12 DVRS1 - " " " " " " (ALM, DCU, IOP)
13 DVOV1 - magtape driver
14 DVOV2 - " starter
15 DVOV3 - " block I/O routines
16 DVOV4 - I/O initializer for MCA
17 DVOV5 - MCA interrupt service
20 RROV1 - page-sec update; check user address for validity; block/unblock; allocate/release core for a process
21 RROV2 - clear a specific area; set termination bit (PBITS); enqueue a process to Core Manager queue; release directory element; emergency shutdown for the file system
22 SYILP - synch. line protocol functions
23 SYCLP - " " "
24 SYRCH - " " receive routines
25 SYTCH - " " transmit "
26 LPBDR - data channel printer driver
27 LPBIO - " " I/O routines
30 DVOV6 - real time clock interrupt handler
31 RROV3 - shared page read
32 DPOV1 - page release for demand paging
33 VDUMP - main dump module
34 VDUM1 - IOP memory dump; DCU dump
35 DUMM0 - empty overlay -- used to hold place in list to prevent renum.
36 DUMM1 - same as DUMM0
37 DZOV1 - Zebra disk driver (interrupt service and device startup)
40 DZOV2 - " " " (init UDB, setup, check timeout; data-lates handling for all devices)
41 DKBDR - paging disk driver (interrupt service, init, setup, enqueue buffer header, map queue table on data channel)
42 IOOV1 - second user I/O overlay : ?PRDB,?PWRB (physical block I/O)
43 IRECD - IPC receive (as a direct call)
44 DRSND - IPC send (as a direct call)

```
45  RESL0 - pathname resolution
46  RESLV -     "         "
47  RESL2 -     "         "       ; link resolution
50  OPEN  - open   a file
51  CLOSE - close  "    "  ; kill FCBs or CCBs
52  DE    - delete "    "
53  CREAT - create "    "
54  DRLSE - release a LDU
55  PROC  - process creation
56  PROC2 -    "         "
57  PROC3 -    "         "
60  PRCNG - process termination; ^C^A processing
61  SSOV3 - pick up/grow/release a swap area; initial program load
62  SSOV4 - get runtime statistics;get username;PSTAT; AMAP
63  SSOV5 - get program name/father PID; get/set searchlist; get a
            console port #; get/set bias; start/kill a histogram
64  DINIT - init LDU
65  XINIT -   "    "  segment 2 & disk init error processing
66  NAMES - pathname resolution
67  CHAIN - chain to new process
70  SSOV2 - change superuser status; ?GHRZ, ?UDIS, ?UEBL, ?INTWT;
            user device support; special shared open for GHOST; ?SINFO
71  SSOV6 - task abort routine for tasks with outstanding receives, on
            the delay chain, doing I/O to MCA or magtape
72  DIRST - set/change/release working directory; CPD size; ?FSTAT
73  ACLST - set/get ACL
74  PNAME - get process name or PID
75  SYSER - system error handler
76  NAME2 - link file processing; rename
77  SSOV7 - system log file support
100 SSOV8 - move bytes to/from user to another
101 SSOV9 - ?MEMI, ?RPAGE with flush, ?FLUSH
102 INFOS - Infos calls to create/read/write a UDA
103 IREC  - IPC receive, call restarted as non-direct
104 ISEND -  "  send     "       "     "   "     "
105 ISEN2 -  "    "  messages spooling
106 CLNUP - process termination clean up
107 IPC   - IPC port look up & translation
110 SOV11 - break file generation; process priority/type change;
            Infos process initialization
111 SOV10 - directory block allocation
112 ECTX  - extended context set up and release
```

```
113 SOV12 - file system shutdown;get CCB;?SATR,?GTACP;searchlist release
114 SOV13 - generate ACL for CREATE; DEFACL; change superprocess
            status; return time; delete IPC entries
115 CONX  - connection management calls
116 ACPRV - establish access privileges for username; powerfail restart
            finish up
117 RESET - get directory element; reset I/O world; block/unblock
            process system call; release shared page
120 NETI1 - network initialization
121 NET1  - miscellaneous networking support routines
122 SRI   - System - REMA Interface module
123 SOV14 - wake up a task
124 SOV15 - set/get system ID; look for a filename in a directory;
            declare a peripheral manager; ?UPDAT system call
125 SOV16 - move bytes to/from shared area
126 SOV17 - ?SPY; address translation for PMGR; keyboard interrupts
            on/off/waiter; get/set shared partition; get user memory map;
            GHOST initialization; ?CTERM (term customer)
127 TARGT - target PID validation for target type calls (?TERM,?BLKPR,
            ?RUNTM,?PRIPR,?PSTAT,?IHIST,etc.); AP support
130 SYOV1 - ?SEBL, ?SDBL
131 SYOV2 - ?SSND, ?SRCV, ?SGES
132 SYOV3 - synch line completions processing
133 SYOV4 - SLM and synch DCU initialization
134 SYOV5 - synch line buffers allocate/check/enqueue/dequeue; ?SDPOL,
            ?SERT, ?SDRT
135 SYOV6 - synch line init/reset/error printing
136 SHOV1 - HDLC synch line support
137 SHOV2 -   "     "      "      "
140 SHOV3 -   "     "      "      "
141 SHOV4 -   "     "      "      "
142 SHOV5 -   "     "      "      "
143 SHOV6 -   "     "      "      "
```

LIB3 - Overlay files in load order
Table 2.2.5

2.3 Image transformations during system initialization

During system initialization, AOS carves the structure of its logical address space out of the initial image of the .SY file followed by empty core. Several events comprise the transformation and they can each be described as some combination of these 4 categories:

a) Core Map Entry changes - a physical page's status may change from being free to being used by AOS in any of several ways (as an overlay, or GSMEM page, etc.). There is a Core Map Entry for every physical page of core and it indicates whether a page is free or its current usage. Core map entries for free pages are linked together. FMCHN is this chain header

b) Physical changes - this occurs whenever a page is written to. For example when an overlay is read from disk, some physical page's contents are changed to contain the overlay.

c) Slot changes - AOS's logical address space contains those physical pages which the "B" map has assigned. When the "B" map is re-loaded, different physical pages may be accessed.

d) GSMEM change - GSMEM memory consists of several chains named FC8, FC16, FC32, FC64, FC128, and FC256. Small blocks of memory are allocated and de-allocated off these chains.

The following events occur during system initialization :

1) Overlay Initialization
The pages occupied by the system overlays (with the exception of the first 2 overlays) are released. The first two overlays are retained because calls will be made to these before the master logical disk unit is initialized. It would be impossible at that time to read an overlay in from disk.

2) Initial release to GSMEM
About halfway through system initialization the memory occupied by part of SINIT which is no longer needed is released to GSMEM space. This is accomplished by calling routine RSMEM in module MEMRY. This must be done because SINIT makes GSMEM calls and there must be some memory on these chains to allocate before the FMCHN is set up (see GSMEM calls below).

3) Overlay calls
When an overlay from disk is required, a page is allocated off FMCHN - which has now been set up - to hold the overlay. It is read in, then the "B" map is reloaded so that the physical page with the overlay in it is in AOS address space.

4) GSMEM calls
When there is a GSMEM request and nothing free on the chains, a page is allocated from FMCHN and placed in AOS's logical address space at the lowest unused GSMEM slot. This page is then split up and the pieces put on the GSMEM chains. Note that this page allocation cannot occur if all the GSMEM slots are in use, i.e. all the way up through the 56,000 slot.

5) RSMEM calls
Conversely, if enough GSMEM areas are released to leave an entire slot unused, that physical page will be returned to the FMCHN and the "B" map will be re-loaded so that the slot is no longer a usable part of AOS's address space.

6) Window mapping
SINIT accesses some data bases (such as FCB's) via window mapping. I.e. once a page has been allocated from FMCHN to hold the FCB's, the "B" map is re-loaded with that page in slot XXX anytime it is necessary to read or write an FCB.

7) System initialization clean up
When SINIT is nearing completion, the pages containing CLIBT and SCALL become a user's process image. This is done by moving a copy of CLIBT to some free pages. These pages are loaded into the "A" map as part of a user's context.
The original copy of CLIBT is split into pieces and placed on GSMEM chains. The "B" map does not need to be re-loaded.
AOS databases necessary to support this initial process are set up and CLIBT becomes a process in its own right. It can now make system calls to proc the PMGR and the operator's CLI before it does a self-term. Prior to the scheduling of CLIBT, SINIT chains to routine INREL in file SSUV1 to release all SINIT memory to GSMEM.

2.4 Logical address space during normal operation

After system initialization, AOS's logical address space has assumed the structure it will use throughout its operation. The 1Kw slots of the address space fall into four major categories, as follows :

a) The writeable first K ("system page 0")
   - contains global variables, constants, and pointers (1 slot)

b) Resident code
   - most of this area is write-protected. Note that there is a small writeable area between the write-protected area and GSMEM space (number of slots dependent on the number of devices the system is driving. This ends following the MEMAP table defined in file MEMAP.SR)

c) GSMEM
   - a dynamic memory pool (number of slots is potentially all those between the resident code and the 60,000 slot at 24.K). GSMEM is allocated on demand, to hold various databases : eight virtual processors, the LCB's, IOCB's, UDB's, system buffers & buffer headers, overlay headers, system CCBs, FCB page headers, CPD descriptors, and other miscellaneous items.

d) window slots
   - the use of the slots is defined for mapping as follows :

```
60000 : quarter pages for PTBL extenders and user CCB's
62000 : CCB's, virtual PTBL's
64000 : FCB's at interrupt level, miscellaneous mapping
66000 : virtual Core Map Entries, GVMEM, misc. mapping
70000 : user parameter packets, miscellaneous mapping
72000 : user parameter packets, FCB's, misc. mapping
74000 : overlays, GHOST page 0 during GHOST entry/exit
76000 : user's page 0
```

There are two hardware maps used by an AOS system. Each map contains 32. slots, one for each K of the 32.K which can be addressed by a program. The "A" map is re-loaded with a user's context each time AOS schedules a different process to run. The "B" map contains the operating system's context. Once set up, the slots below GSMEM are never reassigned and the contents of the map are changed only to modify the amount of GSMEM space or to use a window slot to look at a different physical page. The physical page assignments for the above four categories are :

a) First K
   - slot 0 will always be mapped to page 0 of physical memory

b) Resident code slots
   - AOS slot number X will always be mapped to page X of physical memory. Note that this means that the first slots of the address space (including page 0) are exactly the first part of the .SY file as loaded in core. Which allows code in these slots to execute properly whether or not the map is turned on. This is a necessary feature, since there are times when the system must be able to function in unmapped mode, such as when initially processing an interrupt.

c) GSMEM slots
   - The physical pages used are allocated out of unused core pages and may exist anywhere in physical memory. If only 1 K of GSMEM is needed, all the remaining GSMEM slots will be mapped as invalid. These other GSMEM slots are brought into the address space as needed and declared invalid again when not needed.

d) Window slots
   - The use of window slots (to be able to access more than 32. pages or to look at a page in a user's context) requires that these slots be frequently remapped. The physical pages which are accessed in this way may be anywhere in core.

Logical addr.	Module,library or usage	
0	SZERO	
400	STABLE	
2000	SCHED	
!	DEVLIB.LB	(DEVLIB2.LB)
!	DUMPR	
!	LIB2.LB	(LIB4.LB)
!	DCTLIB.LB	(DCTLIB2.LB)
!	PANIC	
V	MEMAP	
variable	GSMEM space	
60000	window slots -	
77777	up to end of address space	

Address space after system initialization

Table 2.3.1

2.5 Image transformations during normal system operation

Normal AOS operation uses the mechanisms described above for handling overlays, GSMEM requests, and window slots. Page allocation is more complex under normal system operation however. This is because, on a busy system, there may be no free pages on FMCHN. Once the system is up and running processes, any physical page in memory will be found to be used in one of the following ways :

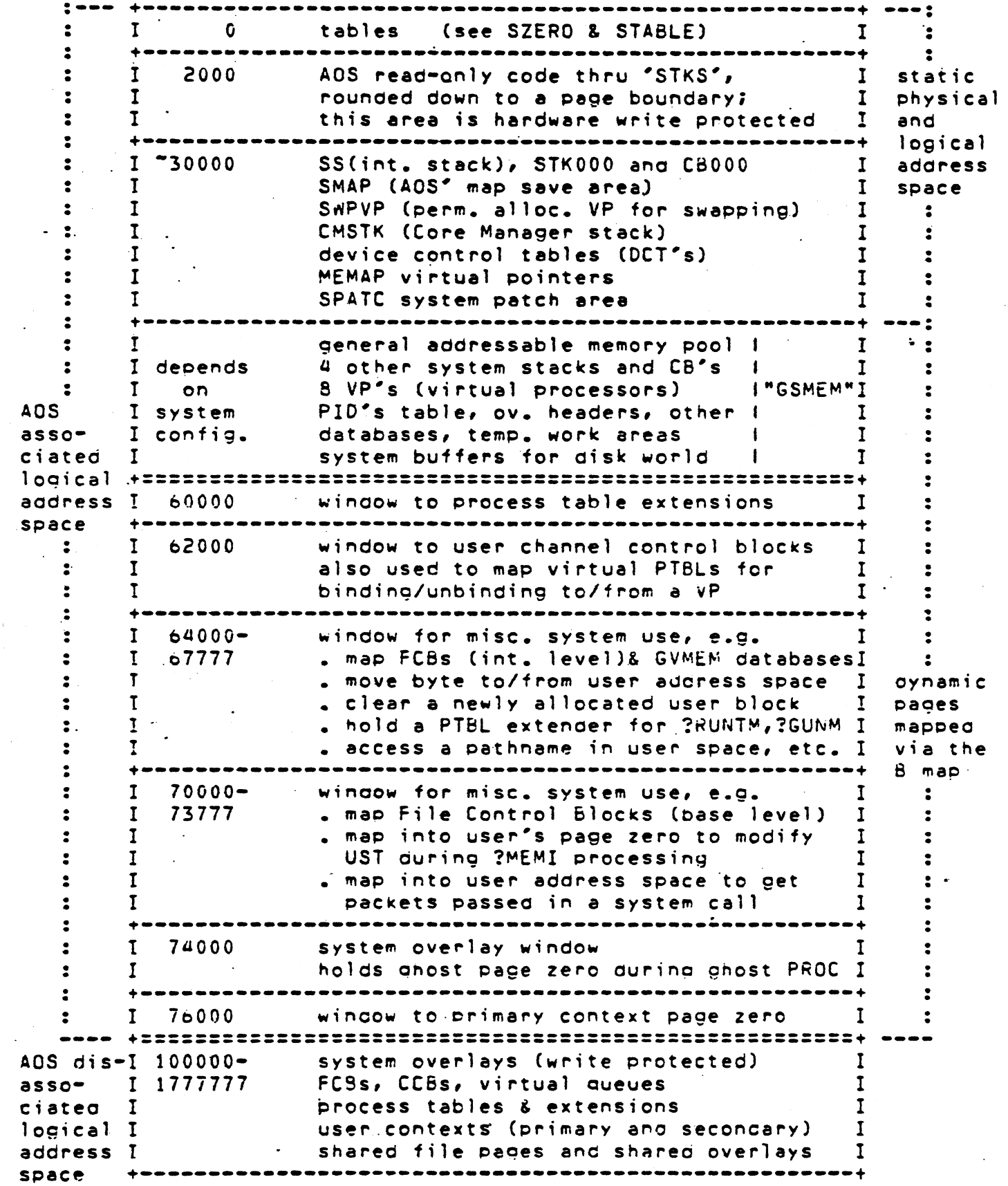
a) free page on FMCHN
b) part of a user's image
c) on the shared candidate chain CANCH
d) in the AOS resident area
e) in GSMEM use
f) in a window slot
g) in disassociated memory (as a data page which may be window-mapped)

If there are no pages on FMCHN, the least recently used shared page on CANCH will be taken. If that chain is also empty, the Core Manager may attempt to swap out (preempt) a process to free up some memory.

2.6 Memory organization

The following diagram is a picture of the physical memory organization under AOS. It summarizes the steps described above and the structure it describes can be found in dumps taken at any time after initialization has completed.

The system logical address space is also referred to as "AOS associated logical address space". The rest of the memory, which AOS cannot address directly, is called "AOS disassociated logical address space" or, more simply, "hyperspace".


0
tables (see SZERO & STABLE)
2000
AOS read-only code thru 'STKS',
rounded down to a page boundary;
this area is hardware write protected
~30000
SS(int. stack), STK000 and CB000
SMAP (AOS' map save area)
SWPVP (perm. alloc. VP for swapping)
CMSTK (Core Manager stack)
device control tables (DCT's)
MEMAP virtual pointers
SPATC system patch area
general addressable memory pool
depends on system config.
4 other system stacks and CB's
8 VP's (virtual processors)
PID's table, ov. headers, other databases, temp. work areas
system buffers for disk world
"GSMEM"
AOS associated logical address space
60000
window to process table extensions
62000
window to user channel control blocks
also used to map virtual PTBLs for binding/unbinding to/from a VP
64000-67777
window for misc. system use, e.g.
. map FCBs (int. level)& GVMEM databases
. move byte to/from user address space
. clear a newly allocated user block
. hold a PTBL extender for ?RUNTM,?GUNM
. access a pathname in user space, etc.
dynamic pages mapped via the 8 map
70000-73777
window for misc. system use, e.g.
. map File Control Blocks (base level)
. map into user's page zero to modify UST during ?MEMI processing
. map into user address space to get packets passed in a system call
74000
system overlay window
holds ghost page zero during ghost PROC
76000
window to primary context page zero
AOS disassociated logical address space
100000-1777777
system overlays (write protected)
FCBs, CCBs, virtual queues
process tables & extensions
user contexts (primary and secondary)
shared file pages and shared overlays

2.6.1 System page zero

The lower page zero is defined as addresses 0-377 (in SZERO) and upper page zero as addresses 400-1777 (in STABLE).

The lower page zero contains constants and pointers for the system :

- auto increment and decrement locations are used for constants, hopefully these will never be addressed indirectly
- portion of this lower page is checksummed, as it is supposed to remain constant. When system is idle, it checksums this area (in SCHED). The section checksummed is from ZCKST to ZCKEN-1, plus locations 20-37 (auto inc/dec locations)
- remainder ZCKEN - 377 is variable data

The upper page zero (400-1777) contains mostly counters and disk pointers. It cannot be checksummed or write protected, because the data here is variable.

SZERO - some important locations :

0-17	defined by hardware (see an "Eclipse principles of operation")
20-37	numeric constants, never referenced indirectly
40-47	defined by hardware
ZREL	map constants for DUA, DUC, LMP, etc., processor dependent map DUA constants for slots 60000 through 76000, set up by SINIT more constants bit pointers to process table status and flag words global entries to the following sources modules: - SGSUB (misc. commonly used subroutines) - SOVLY, ROVCL (system overlay handlers) - STABL (upper page zero tables, and pointers) - MAPER (routines to map system windows) - MEMRY (system memory management routines) - COREM (the core manager) - I/O modules, PANIC, miscellaneous entries system parameters; these include: - the minimum time that a system overlay must remain in memory before its memory can be released, - the number of PIT ticks in a sub-slice (320.), - the size in words of a process table extender (136)

ZCKEN
locations for temporary storage
SYSIN system state flag : 1 => in system , 0 => in user
INTLV interrupt level counter: -1 => at base level, or -1 + # of levels
CUR60 current AOS map slot contents for window at 60000
CUR62 " " " " " " " " 62000
CUR64 " " " " " " " " 64000
CUR66 " " " " " " " " 66000
CUR74 " " " " " " " " 74000
IMDCU DCU word: 0 => no DCU, else IMDCU = the highest data channel slot assigned to the (asynchronous) DCU
CC word pointer to the current control block or virtual processor now receiving CPU attention (loc = 365)
CRSEG current system overlay segment (see chapter 3) else 0 (loc = 366)
DCUWK DCU flag : set to -1 when the DCU or IOP wants to reschedule the peripheral manager (loc = 367); checked by the scheduler

Note : locations CC to DCUWK are absolute, not relocatable. In particular, 'DCUWK' must be at 367 since the DCU modifies this location.

STABLE - some important locations (all addresses are NREL relocatable) :

PIDBT bit table of 16. Words, 1 bit per PID in use
VPTTB 16. (max) page pointers to pages containing process tables
SMAP pointer to AOS's map save area
OVFAH two word disk address to the system overlay file
SWCCB swap file CCB address
RTCCB system root CCB (20 words)
PIFCCB process information file (PIF) CCB address
PERCCB peripheral directory CCB address
CNXTB connection table address
LHNAM local host name area and other networking additions
FNQB pointer to head of free NQ block list
IORUN IOCB running list head
IOFRE IOCB free list head
CCBWQ CCB wait queue head
RLCFL RUNLCB flag - non zero if there is something for RUNLCB to do
CKSUM the checksum computed by the checksum loop
various overlay control tables
system log file database
RESCH, UPQUE reschedule flags
DCT's for powerfail, real time clock, PIT, ERCC option
NERCC number of ERCC errors to correct (256.) before panic
DCHN pointer to delay chain of PTBL's with outstanding ?DELAYs
synch line timer constants
APSIZ size of AP memory or -1 if not defined
APPID the pid to which the AP is assigned
APLOD -1 if AP WCS is not loaded; 0 indicates AP is loaded

```
BTAIL   pointer to the end of the buffer LRU chain
BFLRU   pointer to the beginning of the buffer LRU chain
BFMIN   minimum # of buffers
CACNT   count of allocated cache buffer descriptors
CAHDP   pointer to head of descriptor LRU chain
CATLP   pointer to tail of descriptor LRU chain
FC8 -   pointers to the 'GSMEM' space free memory chains
FC256
VC16 -  pointers to the 'GVMEM' space free memory chains
VC256
FQPG    pointer to the the free quarter page descriptor chain, in
        disassociated memory; this chain is ordered by physical page #
NFQPG   the number of quarter pages on the free chain; there is a minimum
        of five quarter pages in the system
FMCHN   pointer to a chain of free page numbers
CANCH   pointer to a chain of candidate page descriptors
        for shared pages
FCBCH   pointer to the chain of 8 word descriptors for FCB's
BIAS    minimum # of non-interactive processes in memory
ELQUE   pointer to the queue of eligible bound processes
VELQUE  pointer to queue of eligible unbound processes
VELEND  pointer to tail of VELQUE
VPCHN   free virtual processor chain
IESWP   pointer to queue of ineligible swappable processes
IERES   pointer to queue of ineligible preemptible processes
BLKQ    pointer to queue of blocked processes
BLEND   pointer to the last entry on the block queue
RTPTB   dummy process table for the root process
METST -> METST+METLN-1 various metering locations that
                       can be used to look at AOS's current performance
                       characteristics. (See ?SPY below)
PS000   first location in stack pointer table
        each entry points to stack (5 total,1b0=> in use)
MDCH1-  data channel map slot usage table, 1 bit per slot, 32. slots
MDCH8   per map, four maps accessible (A, B, C, D)
ITBL, BTBL   interrupt vector table (see chapter 4)
etc...
```

2.6.2 The ?SPY system call

The ?SPY call maps the system page zero into the calling user's context so that he can reference the performance data kept in STABLE starting at offset METST. SPY retrieves the number of unshared pages in use from the caller's process table, and from this builds a map word to reference the first unused page in the user context as logical address and physical page zero as a page number, write protected! The logical address in the user context where the system page zero is mapped is then returned to the caller's in AC2.

SPY is an entry in overlay SOV17.

2.6.3 ZMAP

AOS uses a null filled 'page' of memory to zero sections of the disk. It does so in the following manor:

If the map that AOS is running on does not contain full memory, and the map will return zeros on an invalid request, then we use a non-existant page in ZMAP.

If the map that AOS is running on contains full memory, or the map will return -1 on an invalid request (currently this happens only on an S-140 or MV-8000), we will allocated the last physical page for ZMAP.

If the system that AOS is running on contains the AP option, then we allocate the last 4 physical pages for AP use and if necessary allocate the fifth last for the diskworld's ZMAP.

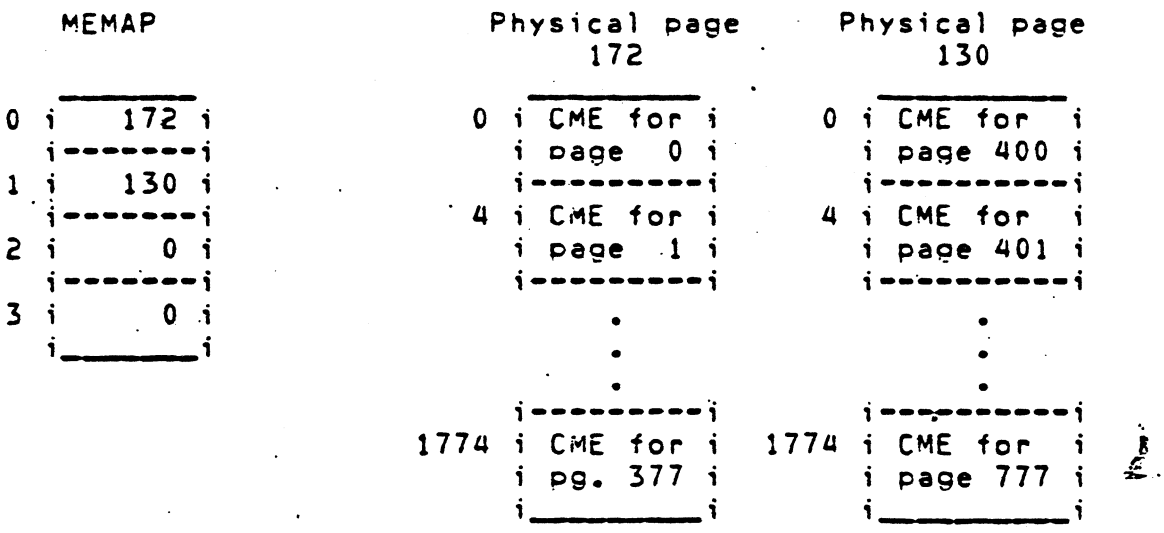
2.7 The AOS Memory Map

The data base AOS uses to keep track of each 1 Kw page of physical memory is defined in module MEMAP.SR. The memory map contains one four-word entry for each physical page in the current memory configuration. These entries are called "Core Map Entries" or "CME's". The memory map - called CBASE in Revision 1 of AOS - used to come out of AOS's dynamic memory area ("GSMEM space"), thus the larger the amount of physical memory, the larger the memory map and the smaller the amount of dynamic memory left. This was more than unfortunate, because it meant a larger memory system would actually be able to support less activity before the dynamic memory became exhausted. Note that in a system with 128 Kw of memory, the memory map requires 4*128 = 512 words, but with 512 Kw of memory this increases to 4*512 = 2048 words or two pages.

This difficulty was solved in Rev. 2 by making the memory map a virtual database which is mapped whenever a reference is made to a CME. Thus now any size physical memory can be supported equally well, with no effect on the amount of dynamical memory available to the AOS.

The system initialization routine SINIT allocates the number of 1 Kw pages necessary to hold the memory map after it has sized memory to determine the amount of physical memory actually present on the system. It fills in a table describing which 1Kw page has been allocated to hold each group of 256. CME's, so that these pages can be mapped as required. This table is called MEMAP and defined in module MEMAP.SR, and MEMAP(I) denotes the page number of the physical page of memory allocated to hold the CME's for pages 0-255., 256.-512., etc.

Thus the structure of MEMAP is (all numbers in octal) :



thus, if N = physical page #, then

N/256. = MEMAP entry defining physical page the CME
and for page N is in
4 * (N mod 256.) = offset of CME within that page.

The maximum length of MEMAP is four, since four pages will hold 1024. CME's, the maximum amount the current Eclipse map will ever permit. Obviously, a MEMAP entry of zero means no page is allocated (physical page 0 can never be allocated) because there is not that much physical memory.

All references to a CME merely call a routine (get Core Map Entry address - GCMEA) that takes the physical page number as input and, doing the necessary mapping, returns a mapped address that can be used to access the CME. The 66000 slot is used to map CME's. Note this slot is not part of a control block's context, thus callers of GCMEA cannot pend and expect the CME to still be mapped after they have been unpended. This is no problem as CME's are typically referenced for only a few instructions.

The CME offsets are defined as follows :

CMPFL forward link if on free chain or candidate chain. Must be offset zero. Termination link value is -1.

CMPBL backward link. Not always used.

CMPST status of this page (left byte). See status bits definition below.

CMPUC use count (right byte)

CMPPT current usage descriptor:
- physical page # if free
- shared memory descriptor pointer if shared
- FCB usage descriptor addr.if used for FCB's
- overlay header addr. if used for overlays
- process table address if given to user
- -1 if allocated to GSMEM or GVMEM space

CMPST status bit definitions :

Label	Bit	Description
CMIOP	0	I/O in progress This bit is set for the duration of a page flush, from the set up of the request to completion. It is checked when looking for a shared page on the candidate chain CANCH, and if set the page is skipped.
CMMOD	1	page has been modified Every time a page is set up as shared without being write-protected, the bit is set. When a shared page is released and this bit is set, the page is flushed back to disk, whether it effectively has been modified or not.
CMSH	2	this is a shared page Whenever a page is set up as shared, this bit is set.
CMERR	3	I/O error detected on page I/O As of 3.11 this bit is never referenced.
CMAOS	4	page in use by AOS As of 3.11 this bit is never referenced.
CMCPG	5	hardware page modified bit On an M600 modbits system, this bit is set when the page actually is modified
CMRZU	6	release to LRU list when count = 0 As of 3.11 this bit is never referenced.
CMUPI	7	call unpend when I/O completes As of 3.11 this bit is never referenced.

The two memory chains FMCHN and CANCH are set up such, that :

- location FMCHN in STABLE holds the page number of the first free page on the free memory chain. The free pages are singly linked through offset CMPFL in the CME, using physical page numbers as links.

- location CANCH in STABLE holds the page number of the first (least recently used) page on the chain of released shared pages. This chain is doubly linked through offsets CMPFL and CMPBL in the CME, using physical page numbers as links.

- location CLEND holds the page number of the last page on the candidate chain CANCH.

2.8 Core Map Entries references

The following is a partial list of system modules that reference CME's, as examples of their use :

CLOSE - reference CME's describing pages of shared files that are being closed.

IOOV - calls SBSRH to search an FCB for a shared page (i.e. to see if a shared page is already in core and so need not be read) and then picks up the shared page descriptor from the CME that SBSRH returns. This CME is needed after a call to SHREL to release the previous contents of the shared slot. IOOV then remaps the CME after the call to SHREL, since SHREL calls RSBLK which maps the CME of the page being released.

MEMRY - this is the primary memory management module in AOS, each routine is described separately (see also paragraph 2.6) :

GSMEM, GSMNW, GSMRS - these three routines allocate dynamic memory space ("GSMEM space") and call GMBLK to get a 1 Kw page. The common code in these routines calls GCMEA which returns the CME address of the allocated page.

GMBLK - this routine allocates a free 1 Kw page to the caller out of FMCHN or CANCH. It returns the physical page number of the allocated page.

RFBLK - this is the complement of GMBLK, and is called to release to the system a free 1 Kw page.

RMBLK, RSBLK - these routines release a shared 1 Kw page of memory to the head and tail of CANCH, respectively. These two routines pass back the mapped Core Map Entry address.

NQCAN - this internal routine actually does the enqueuing of a CME to CANCH.

GCORE - this routine gets the unshared memory for a process by calling GMBLK. It does not itself reference the CME.

SHFLS - this routine flushes a shared page specified by CME address to disk. Since this routine pends while the I/O is in progress, it must remap the CME the caller has specified after the I/O completes so the caller can still reference the CME.

GFCB - This routine allocates FCB's, and calls GMBLK. It then calls GCMEA to get to the CME for the allocated page.

RFCB - This routine releases FCB's, and uses the physical page number in the virtual FCB address to call GCMEA for the page holding the FCB so the FCB header can be found (in offset CMPPT).

SGSUB - routine CHKAR in SGSUB calls GCMEA to compute the CME address of shared pages it wishes to check.

SINIT - SINIT initializes the MEMAP table after it has sized memory. Based on the amount of physical memory found, SINIT will allocate one to four pages to hold the CME's.

SOVLY - the overlay handling module calls GMBLK to get a 1 Kw page to be used for overlays, and then, using GCMEA, updates the CME of the allocated page to reflect its current usage.

SRDB - this is the system shared read code, and this code sets up the CME's for shared pages it reads in, using GCMEA.

SBSRH - this routine searches an FCB shared page list for a given shared page, and returns the mapped CME address of the page if found.

DQCHD - this routine dequeues a CME from CANCH.

STABLE- the definitions of CANCH, CLEND, and FMCHN are documented in this module.

SSOV6 - this module contains a routine called CHKSA which is invoked by an ?SCLOSE if the ?SCLOSE specified that pages of the file being closed be released if necessary. This routine calls GCMEA to compute and map the CME address of each page involved.

SSOV9 - the code in this module that implements the ?FLUSH system call calls GCMEA to compute and map the CME address of each page

SWAPI - the swap in code calls GCMEA to compute the CME address of shared pages of the user context involved.

SWAPO - the swap out code references the CME's of pages released. It calls RSBLK (described earlier) which returns the correct, mapped CME address.

SZERO - entry .GCMEA contains the address of the GCMEA routine.

2.9 AOS memory management

CMEs are used to form chains of full pages of memory (CANCH and FMCHN. A great number of the databases in AOS are much smaller, and therefore AOS must manage smaller chunks of memory in order to avoid waste. The following is a complete list of ways in which AOS manages memory.

1. Full Pages

Full pages are managed by chains of CME discussed above. Again, the chains are:

FMCHN - the chain of free memory pages
CANCH - the chain of shared pages with a use count of 0

2. Virtual Quarter Pages

Virtual quarter pages are used for holding various larger databases (for example Process Table Extenders and user CCBs) Free quarter pages are chained together on FQPG. Allocated quarter pages are pointed to by other databases that reference the database that occupies the quarter page. Virtual quarter page addresses are in the following format:

bits 4-5: quarter page number in physical page (0-3)
bits 6-15: physical page number.

3. GVMEM

GVMEM is the term applied to a pool of various size smaller blocks of virtual memory. The blocks are in 5 sizes and are managed by 5 different chains:

VC16	The	chain	of	free	16	word	chunks	(blocks)
VC32	The	chain	of	free	32	word	chunks	(blocks)
VC64	"	"	"	"	64	"	"	"
VC128	"	"	"	"	128	"	"	"
VC256	"	"	"	"	256	"	"	"

GVMEM is managed by a modified buddy system, which is explained below. GVMEM pointers are in the following format:

bits 0-5: block index (page offset/16)
bits 6-15: physical page number

4. GSMEM

GSMEM is the term used to describe the pool of various size smaller blocks of non-virtual (associated) memory. There are 6 sizes of GSMEM block, and therefore 6 different chains:

FC8	The chain of free 8 word chunks (blocks)
FC16	The chain of free 16 word chunks (blocks)
FC32	etc.
FC64	
FC128	
FC256	

GSMEM is also managed using a modifed buddy system (described below). GSMEM pointers are real addresses.

The Modified Buddy System

GSMEM/GVMEM chains are managed using a modification of the buddy system described in Knuth, "The Art of Computer Programming", vol 1. An explanation, by way of example, as to how it works is as follows:

Assume we need a chunk of 36 words of GSMEM to hold a database. First, we round 36 up to the next multiple of 8, which is 40. We next allocate a chunk of memory from the FC64 chain, passing its address to the routine requiring the memory. Since the database will only occupy the first 40 words, we break the remaining 24 words up into two chunks, one of 16 words, the other of 8 words, and put the addresses of these chunks onto the FC16 and FC8 chains respectively. If there are no chunks on the FC64 chain, we break the first entry on the FC128 chain into two 64 word blocks and put there addresses on the FC64 chain (we now have something on the FC64 chain and can proceed as above). If there are no entries on the FC128 chain, split the FC256 chain and continue. If there are no entries on the FC256 chain, ask AOS for another full page of memory, which we then break into four 256 word chunks and put them on the FC256 chain. If there are no free slots in AOS's GSMEM slots, we either pass back an error to the calling routine if the routine can pend, or ... we are in trouble.

When we are done using the database, we return the memory to GSMEM, and attempt to regroup the block into larger blocks. However, a block can only regroup with the chunk it was split from ... it buddy. The determination of a buddy is done using the following algorithm:

GSMEM block address				size = 8.				buddy address		
130	=	001 011 000	XOR	000 001 000	=	001 010 000	=	120		
120	=	001 010 000	XOR	000 001 000	=	001 011 000	=	130		
110	=	001 001 000	XOR	000 001 000	=	001 000 000	=	100		
100	=	001 000 000	XOR	000 001 000	=	001 001 000	=	110		

GSMEM block address size = 16. buddy address

```
120  =  001 010 000  XOR  000 010 000 = 001 000 000  =  100
100  =  001 000 000  XOR  000 010 000 = 001 010 000  =  120
```

In general:

Buddy address = (block address) XOR (size of block)

Again it is important to stress that a block of GSMEM or GVMEM will only combine with its buddy.

2.10 The Memory Chains

FMCHN - This location holds the physical page number of the page at the head of the chain of free 1Kw pages of memory. The chain is a doubly linked chain of Core Map Entries and uses page numbers as links.

FBLKC - number of pages presently on the free chain.

CANCH - This location holds the physical page number of the first page on the "candidate chain", which is a LRU chain of Core Map Entries describing shared pages that are in core but have a use count of zero. This chain is doubly linked through offsets CMPFL and CMPBL, using page numbers as links.

CLEND - page number of the last item on the candidate chain.

CANCN - number of pages on the candidate chain.

FQPG - This location holds a pointer to the first free quarter page on the free quarter pages chain. This pointer is defined as

```
bits 4-5     quarter page # (0-3)
bits 6-15    physical page #
```

This chain is singly linked using the first word of the quarter page as a link, in the same format, to the next quarter page. The chain terminates with a minus one (-1) in word zero of the last free quarter page on the chain.

NFQPG - number of quarter pages on the free quarter pages chain.

The above locations are all defined in STABLE.

FC8 - There are six consecutive locations in STABLE starting at FC8 and ending at FC256 which contain pointers to the head of six different free memory chains. Each chain consists of different size blocks of free memory in GSMEM space :

FC8	chain of blocks of 8 contiguous words
FC16	chain of blocks of 16 contiguous words
FC32	chain of blocks of 32 contiguous words
FC64	chain of blocks of 64 contiguous words
FC128	chain of blocks of 128 contiguous words
FC256	chain of blocks of 256 contiguous words

FC8 is declared an entry point so that external references to it may be made during assembly and resolved at bind time as well as being able to use FC8 with the debugger. Because of this, references to the other five free memory chain pointers are usually made as offsets from FC8.

The chains are doubly linked through offsets zero and one of the free area, the forward link on each chain being terminated with a minus one and the backward link with bit zero set to 1 and the address of the chain head, i.e. @(FC8 + appropriate offset from FC8).

When all the free memory is used up in GSMEM space the system attempts to expand the GSMEM space by mapping a free page of physical memory into the next open dynamic page slot in AOS system space.

VC - There are six consecutive locations in STABLE starting at VC and ending at VC256 which contain pointers to the head of five different free virtual memory chains, plus a pointer to a mapped chain head. Each chain consists of different size blocks of free memory in GVMEM space. The mapped head is a 66xxx or 67xxx mapped address of a free chain which is presently mapped.

FCH	mapped head address
VC16	virtual chain of free blocks of 16. contiguous words
VC32	virtual chain of free blocks of 32. contiguous words
VC64	virtual chain of free blocks of 64. contiguous words
VC128	virtual chain of free blocks of 128. contiguous words
VC256	virtual chain of free blocks of 256. contiguous words

FCH (also called VC) is declared an entry point and references to the chains are made as offsets from there.

The chains are linked four-way through offsets zero and one, three and four, of the free area. The first two words contain the mapped forward and backward links, the forward link being terminated by minus one and the backward link by bit zero set to one and the address of the chain head, i.e. @(FCH). The third and fourth words are links in the form of virtual pointers, with bits 0-9 holding the physical page number and bits 10-15

the block offset in the page (i.e. word offset/16.). Both virtual forward and back links are terminated with a zero.

When all the free virtual memory chains are used up the system attempts to allocate a free page of physical memory for this use (see the memory allocation routines in chapter 2).

2.11 Memory allocation / deallocation routines

1) GSMEM, GSMNW, GSMRS

- Functional description : get an area of free system memory (AOS' logical address space). The three entry points differ in what happens if no memory is available :

  GSMNW - does not wait, takes error return if free chain empty, without looking at the candidate chain.
  GSMRS - will wait if can flush a page from the shared candidate chain, otherwise restarts the TCB request in order to prevent deadlocks.
  GSMEM - will wait if can flush a page from the shared candidate chain or if a preempt is possible, otherwise restarts the TCB request in order to prevent deadlocks.

- Cautions : the size requested must be in the range 1 to 256. words.

- Error return conditions : no memory available; if GSMEM, also unable to preempt for it.

- Global databases accessed :
  AUSCN = number of 1K pages used by the system
  SDMAP = AOS system map
  Core Map Entries
  Free memory chains FC8 to FC256
  MKEY

- Filename : MEMRY.SR

- Miscellaneous : GSMEM manages the free chains using a modified buddy system (see KNUTH, "The Art of Computer Programming", vol. 1). The size actually allocated is always a multiple of eight, and the area is cleared i.e. set to all zeroes.
  GSMEM will wait if no memory is available, and may preempt another process to get the necessary memory. If AOS is out of address space, i.e. no more map slots are available for GSMEM space, an attempt will be made to release system buffers as long as at least 8 remain.

.2) GVMEM, GVMNW, GVMRS

- Functional description : get an area of free system memory from the virtual memory pool. The three entry points differ in what happens if no memory is available :

  GVMNW - does not wait, takes error return.
  GVMRS - will wait if can flush a page from the shared candidate chain, otherwise restarts the TCB request in order to prevent deadlocks.
  GVMEM - will wait if can flush a page from the shared candidate chain or if a preempt is possible, otherwise restarts the TCB request in order to prevent deadlocks.

- Cautions : the size requested must be in the range 1 to 256. words of memory.

- Error return conditions : no memory available; if GVMEM, also unable to preempt for it.

- Global databases accessed :
  Core Map
  Virtual memory free chains VC16 to VC256
  CUR66 = current contents of 66000 map slot
  MKEY

- Filename : MEMRY.SR

- Miscellaneous : GVMEM manages the virtual free chains using a modified buddy system. The memory size actually allocated is always a multiple of 16. GVMEM returns a virtual pointer to the allocated memory which must first be mapped into the system address space before the area can be accessed. The area is normally mapped into the 66000 slot via the MAPVP function.

  GVMEM shares some code with GSMEM since the functionality is similar. The major differences are that the free chains used by GVMEM are distinct from those used by GSMEM and consequently the enqueuing routines used are different; also the memory allocated by GVMEM is not mapped into the fixed area of system memory, but a virtual pointer to the area is returned to the caller instead, in this format :

  bits 0-5 : block index (page offset/16.)
  bits 6-15 : physical page number

3) GSENQ

- Functional description : enqueue an area of free memory to the appropriate free list.

- Cautions : the size of the area must be a power of two, ranging from 8 to 256.

- Error return conditions : no error return.

- Global databases accessed :
  Free memory chains FC8 to FC256
  Free virtual chains VC16 to VC256
  MKEY = memory key
  VSW on caller's stack = the virtual switch

- Filename : MEMRY.SR

- Miscellaneous : areas are enqueued to the free chains using the first two words of the area as forward and backward links respectively. The free chains are built in ascending order based on the address of the free area, and GSENQ puts the free area in the correct place on the correct chain. If the free area can be recombined with its "buddy" (immediate upper or lower neighbor on the chain), it will be enqueued to the larger size chain, where it might be recombined again, and so on.

  The memory key MKEY will be ISZ'ed when the following two conditions are met :
  - the element is being added to an empty chain
  - all larger element chains are also empty
  which means GSMEM changes significantly when adding a 64 word piece, for example, only if the 64, 128, and 256 chains are all empty.

  If the virtual switch is set, then this routine branches to GVENQ to carry out its function.

4) GVENQ

- Functional description : enqueue an area of mapped memory to the appropriate free virtual chain. The memory address of the area must be mapped in the 66000 slot.

- Cautions : the size of the area to be enqueued must be a power of 2, ranging from 16. to 256. The 66000 map slot is preserved.

- Error return conditions : no error return.

- Global databases accessed : free virtual chains VC16 - VC256

- Filename : MEMRY.SR

- Miscellaneous : areas are enqueued to the free virtual chains using a double set of links maintained in the first four words of the area. The first two words contain the mapped forward and backward links respectively for GVMEM use and are maintained for that purpose. The third and fourth words contain the respective forward and backward links for the chain.
  The actual link words are virtual pointers of the form :

  bits 0- 9 physical page no.
  bits 10-15 block index = page offset/16.

  (Note that this pointer is the inverse of the pointer used elsewhere by mapping functions.)

  Elements of virtual memory are put on the free chains in ascending order based on the value of the virtual pointer to the area. GVENQ enqueues the area to the appropriate virtual chain and assigns the mapped links to point to adjacent areas (if any) contained in the same physical page.

5) GSDEQ

- Functional description : get an area of free memory from the appropriate free list.

- Cautions : the size of the area requested must be a power of two, ranging from 8 to 256.

- Error return conditions : no area of the requested size available.

- Global databases accessed :
  Free memory chains FC8 to FC256
  Free virtual chains VC16 to VC256
  VSW (on caller's stack) = the virtual switch

- Filename : MEMRY.SR

- Miscellaneous : areas are dequeued from the free chains using the first two words of the area as forward and backward links respectively. GSDEQ removes the first area (if any) from the relevant free chain for the size requested. If the virtual switch is set then this routine branches to GVDEQ to carry out its function.

6) GVDEQ

- Functional description : get an area of mapped memory from the appropriate free virtual chain.

- Cautions : the size of the area requested must be a power of two, ranging from 16. to 256.
The 66000 slot may be remapped.

- Error return conditions : no area of the requested size available.

- Global databases accessed : free virtual chains VC16 - VC256

- Filename : MEMRY.SR

- Miscellaneous : areas are dequeued from the free virtual chains using a double set of links maintained in the first four words of the area. The first two words contain the mapped forward and backward links respectively for GVMEM use and are maintained for that purpose. The third and fourth words contain the respective forward and backward links which serve as the actual links for the chain. The actual link words are virtual pointers of the form :

  bits 0- 9 physical page no.
  bits 10-15 block index = page offset/16.

GVDEQ removes the first area (if any) from the virtual chain of appropriate size and recomputes the mapped links of any adjacent areas contained in the same physical page.

7) RSMEM

- Functional description : release a piece of system memory

- Cautions : the size of the area must be in the range 1 to 256.

- Error return conditions : none

- Global databases accessed :
AUSCN = count of 1K pages used by AOS
SMAP = AOS system map

- Filename : MEMRY.SR

- Miscellaneous : the size may be any number within the range specified above, but the true size of the area is assumed to be a multiple of 8. Thus the size is rounded up to the next multiple of 8 if necessary.

8) RVMEM

- Functional description : release an area of system virtual memory. The input virtual pointer must have been created by GVMEM/GVMRS/GVMNW when the area was originally allocated.

- Cautions : the size must be in range 1 to 256.

- Error return conditions : none

- Global databases accessed :
  virtual memory free chains VC16 to VC256
  CUR66 = current contents of 66000 map slot

- Filename : MEMRY.SR

- Miscellaneous : the size may be any number within the range specified above, but the true size of the area is assumed to be a multiple of 16. Thus the size is rounded up to the next multiple of 16 if necessary. RVMEM shares code with RSMEM since the functionality is nearly identical. Notable differences are that RVMEM releases an area of system memory to the virtual system memory pool, and that the enqueuing routine used by RVMEM is distinct from that used by RSMEM.

9) GMBLK/GMCNB

- Functional description : allocate a 1K page of memory. GMCNB will always allocate from the candidate chain.

- Cautions : uses 66000 slot

- Error return conditions : no memory available on free chain or candidate chain.

- Global databases accessed :
  FMCHN = free memory chain
  FBLKC = # of 1K pages on FMCHN
  Core Map Entry of page allocated
  CANCH = candidate chain of useable 1K pages
  FCB list of shared pages

- Filename : MEMRY.SR

- Miscellaneous : a free 1K page is taken from the free list if possible, the candidate chain being looked at only if FMCHN is empty. If CANCH is empty, the Core Manager is awakened. If a page is found on CANCH, it must be dequeued from the chain and from its FCB and written out if modified.

10) RFBLK

- Functional description : release a 1K page to the free chain
- Caution : uses 66000 map slot.
- Error return conditions : none
- Global databases accessed :
  FMCHN = chain of free 1K pages
  FBLKC = # of pages on FMCHN
- Filename : MEMRY.SR
- Miscellaneous : the input page # is AND'ed with PMASK, so that only the relevant bits are looked at (9-15 in 200 map, 6-15 in 400 map).

11) RMBLK/RSBLK

- Functional description : RMBLK/RSBLK release a shared 1Kw page of memory.
- Cautions : RMBLK/RSBLK assume the page being released is shared and is not on the LRU candidate chain.
  Uses 66000 slot.
- Error return conditions : no error return.
- Global databases accessed :
  CANCH = LRU candidate chain
  CANCN = count of pages on CANCH
  CLEND = pointer to end of CANCH
  Core Map Entry of page being released
  SHUCN = count of shared pages in use
- Filename : MEMRY.SR
- Miscellaneous : If the result of the page release is the use count of the page becoming zero, the page is placed on the LRU candidate chain (CANCH).
  If the maximum candidate chain size is exceeded, a page will be popped off and put on the free chain. This is to correct the problem caused by shared page headers eating up GVMEM.
  RMBLK will put the page on the head of the chain (i.e. least recently used), RSBLK puts the page on the end of the chain (most recently used).

12) GQPG/GQPNW

- Functional description : GQPG/GQPNW allocate a quarter page to the caller if one is available. GQPG will preempt if no memory is available, and restart the request. GQPNW will not wait if no memory is available for quarter pages but will return to the caller without waiting for a preempt.

- Cautions : may cause preemption of another process. Uses 66000 map slot.

- Error return conditions : no memory available

- Global databases accessed :
  FQPG = free quarter pages chain
  NFQPG = number of quarter pages on that chain

- Filename : MEMRY.SR

- Miscellaneous : the virtual quarter page address is returned in AC1 as follows

  bits 4-5 : quarter page number (0-3)
  bits 6-15 : physical page number

13) RQPG

- Functional description : release a quarter page back to the system.

- Cautions : RQPG assumes input argument is a valid quarter page address that has been previously allocated via GQPG/GQPNW. Uses 66000 map slot.

- Error return conditions : none

- Global databases accessed :
  FQPG = chain of free quarter pages
  NFQPG = number of quarter pages on FQPG

- Filename : MEMRY.SR

- Miscellaneous : the quarter page chain FQPG is ordered by increasing quarter page address so that 1K pages used for quarter pages can be returned to the system. An internal constant, MINQP, defines the minimum number of free quarter pages to be kept on the chain. If NFQPG > MINQP after the quarter page has been inserted in the chain, an attempt will be made to find four quarter pages from the same 1K page and return that page.

13) SHFLS

- Functional description : flush (write to disk) a shared page
- Cautions : none
- Error return conditions : none
- Global databases accessed :
  Core Map Entry for the page being flushed
  Buffer Header for the shared page
- Filename : MEMRY.SR
- Miscellaneous : this routine is called by GMBLK to flush out the page selected from the candidate chain. If a non modbits system, the shared page is always written to disk. If this is a modbits system, then the page is written to disk only if the hardware page modified bit has been set. if I/O is really necessary, as this has been done before by the caller of SHFLS. The routine does not return to its caller until the I/O has completed.

2.12 Slot mapping routines

All of the routines described below use a common stack template. Each entry processes the type call, then generates a common stack definition of what mapping is required. Control then goes to a common map routine (MAPER) that uses the stack template and handles possible error conditions.

1) MAPVP

- Functional description : map an area of virtual memory into the system memory window.
- Cautions : the area is mapped into the 66000 slot. This slot is not preserved if the system path becomes pended.
- Error return conditions : none
- Global databases accessed :
  CUR66 = current contents of 66000 map slot
- Filename : MAPER.SR
- Miscellaneous : a check is made to see if the area is currently mapped in the system window. If it is then the remap is bypassed.

2) MBL(60,62,64,66,70,72,74,76)

- Functional description : map one page of a user address space into the designated slot of the system address space.
- Assumptions : CC (loc 365) points to a control block that has a process table and TCB address.
- Global databases accessed : control block
- Filename : MAPER.SR

3) IMB(64,66,70)

- Functional description : map one page of a user address space into the designated slot of the system address space. These routines are used when CC does not point to a control block. Hence the process table and TCB address must be passed in explicitly.
- Assumptions : if the ghost flag is set, the call is from the ghost.
- Cautions : the TCB address must be the last temporary on the caller's stack.
- Filename : MAPER.SR

4) MAPP

- Functional description : maps the process table extender of a specified process into the 60000 map slot.
- Assumptions : offset PVEXT in the process table has the virtual quarter page address of the extender to be mapped.
- Cautions : changes contents of map slot 60000, and if a control block is running at the base level, offset CWN60 in the control block so that proper context is restored whenever the control block executes.
- Error return conditions : none
- Global databases accessed :
  process table
  control block (if one running & at base level)
  CUR60 = current contents of map slot 60000
  INTLV = interrupt level indicator

- Filename : MAPER.SR

- Miscellaneous : MAPP may be called from the interrupt world, since the interrupt code saves and restores slot 60000. If called from the interrupt world, CWN60 in the running control block is not changed, since that control block is not making the call and to do so would result in the control block having the wrong context.

5) MAPC

- Functional description : maps a Channel Control Block (CCB) into the 62000 slot.

- Cautions : input values are not checked. Changes contents of map slot 62000 and offset CWN62 in control block if one is running at base level.

- Error return conditions : none

- Global databases accessed :
  CUR62 = current contents of 62000 map slot
  Control block if one running & at base level
  INTLV = interrupt level indicator

- Filename : MAPER.SR

- Miscellaneous : MAPC can be called from the interrupt world since the interrupt code saves and restores the 62000 slot. In this case the running control block, if any, is not updated since it did not make the call and to do so would leave it with an incorrect context.

6) MAPE

- Functional description : maps the designated quarter page into the 64000 slot and calculates the mapped address of the ECB (Extended Context Block).

- Cautions : the 64000 (and 66000) slots are preserved over interrupts but not if the system path becomes pended. MAPE destroys the 64000 slot.

- Filename : MAPER.SR

- Note : the concept of "Extended Context", allowing a resident type process to access pages of memory beyond its logical address space, is not used anymore.

7) DM64/RDM64

- Functional description : these routines map two pages (if needed) of a user address space into the 64000 and 66000 slots of the system address space.
- Assumptions : CC points to a control block that has a process table and TCB address.
- Global databases accessed : control block
- Filename : MAPER.SR

8) ID64/RID64

- Functional description : these routines map two pages (if needed) of a user address space into the 64000 and 66000 slots of the system address space. They are used when CC does not point to a control block. Hence the process table and TCB address must be passed in explicitly.
- Assumptions : if the ghost flag is set, the call is from the ghost.
- Cautions : TCB address must be the last temporary on the caller's stack
- Filename : MAPER.SR

9) MP66

- Functional description : map a virtual quarter page to slot 66000
- Error return conditions : none
- Filename : MAPER.SR

10) DM70/RDM70

- Functional description : these routines map two pages (if needed) of a user address space into the 70000 and 72000 slots of the system address space.
- Assumptions : if the ghost flag is set, the call is from the ghost.

- Cautions : the TCB address must be the last temporary on the caller's stack. This routine will not update CWN70 in the control block, because some callers (IOOV) do not have one. Hence, if a caller with a control block calls this routine and then pends, he cannot assume that the 70000 slot will be restored when he unpends.

- Filename : MAPER.SR

11) RST62

- Functional description : restore the current 62000 map slot. Because of virtual process tables the 62000 slot is often changed by a routine accessing virtual process information. This routine provides a way of restoring the slot.

- Cautions : may modify the current 62000 slot saved in the associated control block, if any.

- Error return conditions : none

- Global databases accessed :
  CUR62 = current contents of 62000 map slot
  INTLV = interrupt level indicator
  CC = current control block
  Control block offset CWN62 = contents of 62000 map slot

- Filename : MAPER.SR

12) IMPF

- Functional description : map an FCB into slot 64000

- Error return conditions : none

- Global databases accessed : CUR64

- Filename : SGSUB.SR

The following mapping routines have special uses :

SWAMP	:	mapping for normal or high speed data channel I/O
ALD	:	used for loading the A map (user contexts)
BLD	:	" " " " B " (AOS context)
DLD	:	" " " " data channel map 1 or 2
ADLD	:	" " " " " " " 1
BDLD	:	" " " " " " " 2
CDLD	:	" " " " " " " 3
EDLD	:	" " " " " " " 4

In summary :

Database	Slot	Mapped by
Quarter page (extend.+map)	60000	MAPP
CCB's	62000	MAPC
Virtual Process Tables	62000	RST62
ECB's (not used)	64000	MAPE
FCB's at interrupt level	64000	IMPF
Quarter pages (general)	66000	MP66
PIB's	64000 & 66000	BLD
GVME^{M} areas	66000	MAPVP
Line Table	64,66,70,72000	BLD
User Parameter Packet	70000 & 72000	DM70/RDM70,IMB70
FCB's at base level	72000	BLD
Overlays	74000	BLD
User Page Zero	76000	MBL76
Random remapping	64,66,72000	

Note : 74000, normally used only by overlays, is also used to map Ghost page zero to only on entering or exiting the Ghost.

Window Map Utilization

CHAPTER 3 - DATABASES, DATA STRUCTURES, DATA RESOURCES (updated for AOS Rev. 3.11)

3.1 Introduction

This chapter will describe data objects and the routines which exist primarily to deal with them, such as creating or deleting them. Data objects are grouped into three categories to better show their interrelationships. Some are databases, such as Control blocks and Process Tables. Each offset in a database has a predefined meaning. These offsets and their symbols are defined in the parameter files PARU, PARS and PARFS. Other data objects may be data resources. These are units of memory (in core, or on disk) which may be allocated to hold a database or some other part of a context. Finally there are data structures, such as the eligible queue ELGUE. Data structures consist of linked databases or data resources.

3.2 Databases

What follows is a description of only a small part of all AOS databases. Namely process tables and their related definitions, because their design has been reviewed in the course of AOS evolution. The Memory Map described in chapter 2 could have been placed here too, and the File System related databases (chapter 5) also...

For an exhaustive list of AOS databases and their offsets definitions, one can always refer to the system parameter files and to the system page zero set up code in SZERO and STABLE.

3.2.1 Virtual process tables

A virtual process table is a process table that must be mapped in order to be referenced, i.e. does not reside permanently in AOS's logical address space. The implementation presented here also added a second level queue to the AOS scheduler, allowing a clean separation of the issues of "eligibility" - allocating main memory to a process - and "runnability" - allocating the CPU to a process - . Thus it helps simplify management of the two most important system resources.

The design also introduced the concept of a "virtual processor" to AOS. A virtual processor is an abstract representation of a physical central processing unit; it is in this sense analogous to a process, which is an abstraction of a user's program. And combining the virtual processor idea with the second level scheduler queue alleviated the problem of AOS logical address space becoming exhausted and the poor performance exhibited in AOS systems with large amounts of physical memory.

One notion introduced in the document "Virtual Process Table Functional Specification" has been dropped, however - the idea of swapped process tables. Swapping process tables to disk entails too many inefficiencies and complications to be feasible. The chief difficulty is referencing process tables when swapped. Any code path that references the process table of a process other than the caller potentially may reference (and modify) a swapped process table. Handling queues of process tables is especially difficult and time consuming if the process tables on the queue can be on disk. Thus this aspect was not included in the final design. This also greatly simplified the implementation.

The most important goal of the virtual process table design was to achieve an implementation of AOS that would fully support 64. or even a larger number of processes in a working environment. The second primary goal of this project was relieving the performance bottleneck AOS encountered when large numbers of processes were active on large memory systems. This problem manifested itself under Rev. 1 when AOS tried to fit as many processes into main memory as possible, with the result that many processes were on the scheduler's eligible queue. The scheduler overhead was then such as the system was spending much of its time deciding what to do! Response and performance should now gradually degrade as load increases, rather than dropping off suddenly when a certain threshold is reached.

The name "virtual process tables" means that process tables are not allocated in GSMEM space, but in hyperspace instead.

In order to be run by the scheduler, a process will have to be bound to a virtual processor. Conceptually the scheduler merely chooses a virtual processor to be run on the physical processor. Pragmatically, a virtual processor can be thought of as a resident process table. By controlling the number of virtual processors (i.e., resident process tables), contention for the physical processor can be reduced.

A pool of virtual processors is allocated during system initialization. The number of virtual processors could be a function of many variables, both dynamic and static, like memory size, number and type of processes on the system, etc. Note this pool could be grown and shrunk dynamically to meet changes in the process environment.

Initially and to keep things simple, a fixed number of eight virtual processors has been implemented. Once some experience concerning the performance of AOS with virtual process tables is gained, evaluation of this algorithm and experiments with other methods including those that dynamically vary the number of virtual processors should be done.

The scheduler deals only with virtual processors, i.e. process tables that have been bound to resident process tables. All other process queues, however, are queues of virtual process tables. The Virtual Eligible Queue (VELQUE) consists of those processes that have been allocated main memory (i.e. are "eligible" by the current AOS definition) but have not been bound to a virtual processor. The virtual eligible queue acts as follows with respect to swapping operations : the Core Manager will select processes from it to be swapped out, and will put processes on VELQUE from the ineligible queues when they have been swapped in.

Hence, the term "eligible process" means a process on the scheduler's eligible queue. An eligible process has a resident process table and has memory allocated to hold its core image. If either of these conditions is not met, the process is ineligible.

Movement from the virtual eligible queue to the eligible queue is controlled by the scheduler. Essentially, when the scheduler puts a process on VELQUE, it means AOS has the memory resource to run the process but not the processor resource. There are many such instances. However, for efficiency reasons, we will never put a process on VELQUE unless there is a process on VELQUE that can be moved to ELQUE. This simply means that if there are fewer active processes than virtual processors, those processes will remain on ELQUE even if they could be made virtual. Processes can be moved between VELQUE and ELQUE at the end of a time sub-slice (see chapter 4).

The major implications of the Virtual Processors design are that the process table address does not remain constant since the same process can be bound to different virtual processors at different times. Hence all references to processes, such as the pointer to the father process, must use the PID (process identifier) rather than a process table address. All queues of processes such as the delay-chain, the histogram chain, etc. as well as the various scheduler queues are virtual; except, of course, ELQUE.

3.2.1.1 Some Process Table Offsets

PDAD, PSON, PSONL

These three offsets link process tables according to the process hierarchy : PDAD links a process to its father process, PSONP to one of its sons, the rest of the sons are linked to the son pointed to by PSONP through PSONL. These three chains use PID's as links.

PFNVT

This status bit is defined in flag word PFLG4, and means "Do not make this process virtual". Thus only processes bound to virtual processors can have this bit set. If set, the process cannot be unbound until the bit is cleared. This is necessary, for example, so that a process with a system call in progress does not get unbound, as the process table address in the control block processing the call (offset CPTAB) would no longer be valid. Note this particular case could be handled by checking PSQCT, the process table extender offset with the number of outstanding system calls, but it is better to consolidate the special checks into one bit. PFNVT must be set whenever PSQCT is incremented, and must be cleared whenever PSQCT is decremented to zero. This is done by SCPRC and the various I/O post processors. If the process is the PMGR, PFNVT will be set always, for efficiency.

PFBVP

This status bit is also defined in flag word PFLG4, and if set indicates the process is bound to a virtual processor. This is mostly for ease of implementation, as there are cases when it is necessary to remove a process from a queue, and the state of this bit determines whether the process is on the eligible queue or one of the virtual queues. Thus PFBVP is set if and only if the process is bound to a virtual processor and is on the eligible queue.

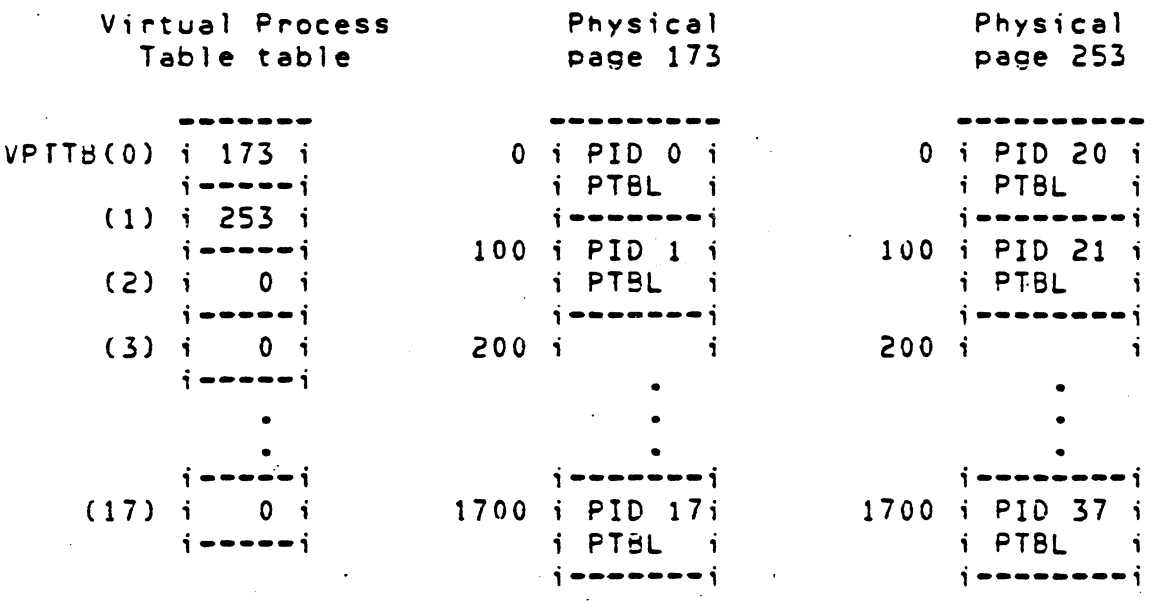
PSSEL

This offset contains the number of sub-slices the process has used since it was last put on the eligible queue, i.e. bound to a virtual processor. PSSEL is zeroed each time the process is bound, and incremented each time a sub-slice expires. But this offset is never really used and never checked by the scheduling algorithm.

3.2.2 VPTTB - The Virtual Process Table Table

VPTTB is a data base defined in STABLE that records the mapped memory used to hold the virtual process tables. It contains one entry for each physical page of memory allocated to hold virtual process tables. Note that since the resident portion of the process table is less than 64. words (100 octal), 16. virtual process tables will fit in one page of memory (64. words per process table * 16. process tables = 1024. words). Since AOS limits the number of PID's to 256. (ranging from 0 to 255.), the maximum length of VPTTB is 16. words and the structure of VPTTB is as follows :

VPTTB(I) = physical page of memory allocated to hold the process tables for processes with PID's in range 16.*I <= PID < 16.*(I+1)



Let N = integer quotient of PID divided by 16.
let M = remainder,
i.e. PID = 16.* N + M

then :

VPTTb(N) points to physical memory page holding virtual process table for the process

and

100 * M = beginning offset of the process table within that page

3.2.3 PIDBT - The PID's Bit Table

PIDBT is a bit table containing a bit for each possible PID. Bit number I will be zero if no process currently exists with PID = I. Bits 0 and 3 are preallocated for the root process and for CLIBT.

3.2.4 PIDTB - The Table of PID's

PIDTB is the process-identifier-to-process-table-address conversion table. Its current size is kept in location PIDLN in page zero. PIDTB entries are defined as follows :

PIDTB(PID) = 0 if no process exists with that PID

PIDTB(PID) = 100000 if the process with that PID exists but is not bound to a virtual processor (i.e. has a virtual process table only)

PIDTB(PID) = virtual processor address if that PID exists and has been bound to a virtual processor (i.e. has been allocated a process table in the system address space).

Thus, if PIDTB(PID) = 100000, the process table can be accessed by using VPTTB as described above to find the appropriate physical page and mapping it.

Pages are allocated to hold virtual process tables only as needed. PROC2 expands PIDTB when necessary and allocates another page to hold virtual process tables, filling in VPTTB accordingly. Since initially PIDLN = 16., one page will be allocated and one entry in VPTTB filled in. This is done by SINIT.

As of 3.11, PROC2 limits the highest PID to 100, hence PIDTB will have a maximum length of 120 (80.) to accommodate the 65. possible PIDs. PIDBT and VPTTB are assembled in to allow 256. PIDs.

Note the above means VPTTB(I) = 0 if 16.*I > value of PIDLN .

3.2.5 Control blocks

Control blocks are used for cases in which the system needs a stack to handle a code path, and there is a possibility that the path will pend. These stacks may be allocated when a user makes a system call which requires a stack, or they may be allocated for system use.

There are 5 standard control blocks and one special CB called the core manager task (CMTSK). In chapter 4, we will discuss what the core manager actually does. Each control block has its own stack and these will be discussed later when we deal with the subject of stacks.

The maximum number of standard control blocks is controlled by SCNCB, defined in PARS. The number of remaining standard control block stacks not currently in use by the system is kept at location STKCT.

3.3 The major AOS scheduling queues

NOTE: Queues used by the AOS diskworld are discussed in chapter 5.

AOS maintains 5 major queues that are used in scheduling and scheduling related functions. These are:

ELQUE	the eligible queue
VELQUE	the virtual eligible queue
BLKQ	the blocked queue
IESWP	the ineligible swapped queue
IERES	the ineligible resident/preemptible queue

ELQUE is a special queue for two reasons. One, it is the only queue which contains databases that are in AOS's associated address space. Two, it is the only queue that contains both process tables and control blocks.

All of the other queues reside in AOS's disassociated space and contain only process tables.

There are two offsets found in both the process table and the control block which are used to link items on the queues together, they are:

PLNK	which is the link to the next item on the queue.
PBLNK	which is the link to the previous item on the queue.

When the item is on the eligible queue, the link is the actual address of the the next item (since all entities on the queue are directly addressable). When the process table is on any of the other queues, the link is the PID number of the next process table on the queue.

These queues (due to the common link word) are mutually exclusive, implying that a process can only be on one of the queues at a given time.

3.3.1 ELQUE - The Eligible Queue

ELQUE - This location, defined in STABLE, points to the head of the eligible queue, which is always the core manager control block address.

The eligible queue is a linked list containing the Core Manager, any control blocks in use, and the process tables of any eligible processes which have been bound to a virtual processor. Subsequent links in the chain are linked both forward and backward through offsets in the process table, and all links are either control block or resident process table (i.e. virtual processor) addresses.

The eligible queue is always headed by the Core Manager control block, and the last entry on the eligible queue is the address of the root process table. The order of entries on the queue is determined by the entries PNQF which will be discussed later.

PLNK - The forward link offset. Currently set to five (5). The forward link of the last process table on this chain contains a minus one (-1).

PBLNK - the back link offset. Currently set to six (6). The back link of the last process table on this chain has bit zero set to one (1B0) and contains the address ELQUE-PLNK. Thus the terminating back link plus the forward link offset can be used to get the head of the eligible queue, i.e. last back link = @(ELQUE-PLNK), and this is found in the Core Manager control block at offset 6.

ELINT - number of eligible interactive processes.

ELNON - number of eligible non-interactive processes.

These two counts are defined in STABLE. For AOS, a non-interactive process is a swappable process having a time slice exponent of 6 (see chapter 4.5.2). An interactive process is any other swappable process.

3.3.2 VELQUE - The Virtual Eligible Queue

VELQU - This location defined in STABLE contains the PID of the first process on the virtual eligible queue. All processes on VELQUE are "eligible", that is their process images are core resident (more simply put, bit PFELG = 1). They are all candidates to be bound to a virtual processor and placed on ELQUE, but conversely they are also candidates for swapping (unless their

process type is resident - this design does allow resident type processes to be unbound from virtual processors).

As is ELQUE, VELQUE is prioritized by PNQF (see chapter 4). Like the other scheduler process queues, VELQUE is doubly linked, using offsets PLNK and PBLNK, which contain the PID of the next and the previous processes on the queue, respectively. The forward link will terminate with -1, the back link with 1B0 + address of VELQUE.

VELEN - this location defined in STABLE contains the PID of the last process on VELQUE.

3.3.3 BLKQ - The Blocked Queue

BLKQ - This location, defined in STABLE, contains the PID of the process at the head of the blocked queue.

The blocked queue is a linked list of the process tables of all the processes that are currently blocked. The chain is linked both forward and backward through offsets PLNK and PBLNK of the process tables in the queue, using PIDs.

The forward link is terminated by a minus one (-1) and the back link is terminated by bit zero set to one (1B0) and contains the address of BLKQ-PLNK, i.e. @(BLKQ-PLNK). This allows the Core Manager not to have to treat the end link as a special case.

All processes that have been blocked will be on this queue. It is a queue of virtual process tables, which implies the act of blocking a process should unbind the process from its virtual processor if it is bound to one. But it may not always be possible to unbind the process (for example if a system call is in progress), so the scheduler will keep trying to unbind processes it finds on ELQUE that are blocked, and move them to BLKQ.

Processes on BLKQ may still have memory allocated to them.

The following locations are defined in STABLE :

BLKCT - number of blocked processes

SRBLC - number of resident blocked processes

BLEND - PID of the last process on the blocked queue

BSFLG - scan inhibit flag. Non-zero value prevents scheduler from scanning this queue.

3.3.4 IERES - The Ineligible Resident Queue

IERES - This location, defined in STABLE, contains the PID of the process at the head of the ineligible resident queue. This queue links the virtual process tables of any resident or preemptible type processes that have not been allocated memory. Resident type processes can only be on this queue when being created, or if they have just had their type changed to resident. Preemptible and resident processes that are blocked will be on BLKQ, not IERES.

All processes on the queue are linked through offsets PLNK and PBLNK in their respective process tables. The forward link is terminated by a minus one (-1), and the back link by bit zero set to one (1B0). PID's are used as links.

Because a process on this queue is swapped out, the process table extender and user status information in user's page zero are not in memory, as well as all the unshared portions of the user's process.

IELRS - number of presently ineligible resident processes

IESFL - when non-zero, inhibits the scan of this queue.

These locations are defined in STABLE.

3.3.5 IESWP - The Ineligible Swappable Queue

IESWP - This location, defined in STABLE, contains the PID of the process at the head of the ineligible swappable queue. The ineligible swappable queue is a doubly linked list of all the swappable processes which are no longer memory resident.

All processes on the queue are linked through offsets PLNK and PBLNK in their respective process tables. The forward link is terminated by a minus one (-1), and the back link is terminated by bit zero set to one (1B0). PID's are used as links.

Because a process on this queue is swapped out, the process table extender and user status information in user's page zero are not in memory, as well as all the unshared portions of the user's process.

Note that blocked, swapped processes will be on BLKQ, not IESWP.

IELSW - number of presently ineligible swappable processes

IESFL - when non-zero, inhibits the scan of this queue.

These locations are defined in STABLE.

Note : AOS maintains two different queues of swapped processes, in order to give resident and preemptible processes preference when contending for memory. When awake, the Core manager will try and fit in a process from IERES, and only if this queue is empty does he scan IESWP.

3.4 The minor AOS scheduling queues

NOTE: Queues associated with the disk world are discussed in chapter 5.

There are three additional queues not discussed in the previous section that deal with processes or scheduling. These are:

DCHN	the chain of processes with outstanding delays
HISLS	the chain of processes with outstanding histograms
CMGWD	the chain of process that are waiting to be swapped (either in or out)

Unlike the case of the major queues, a process table can be on more than one of the minor queues at a given time. However, the process table must also be on one of the major queues if it is on a minor queue.

Unlike the major queues, the minor queues are not linked through PLNK and PBLNK (these are still used to link on the major queue). In the case of the DCHN queue, links are through offset PDLNK of the process table. In the case of the HISLS chain, links are through offset PHLNK of the process table. For the CMGWD chain, the link is offset PCMLK.

3.4.1 DCHN - The Delay Chain

DCHN - This location, defined in STABLE, holds the PID of the process at the head of the delay chain.

The delay chain is a queue used for keeping track of all of the tasks in the system currently doing a delay. A delay is the method by which a user task can send for a specified time without tying up system resources.

The delay chain is actually two linked lists in one. The first one is pointed to by DCHN and is the linked list of process tables associated with processes that have one or more tasks doing delays. The second is a linked list of the process tables in the delay chain, including links to all the tasks in this given process that are currently doing delays.

The process tables in the delay chain queue are singly linked using offset PDLNK in the process table. The linked list is terminated by a minus one in the link word. Note that process tables found on the delay chain may also be found on various other chains such as the eligible queue or blocked queue.

Delaying tasks TCB's for each process are singly linked through offset ?TSYS of the TCB. The list is terminated by a minus one. The TCB's of each process are ordered by the amount of time left to delay. Those tasks with the least amount of time left to delay appear earlier on the chain. If two tasks delay for the same amount of time, the link word ?TSYS of the first task will have bit zero set (1B0).

Note the links are all PID's, even though some processes on DCHN might be bound to virtual processes and others might not.

The following five offsets are defined as process table offsets :

PDLNK - forward process link offset.

PDINH - number of real time clock ticks the first task is to delay, high.

PDINL - number of real time clock ticks the first task is to delay, low.

PDTTH - total delay time on queue, high.

PDTTL - total delay time on queue, low.

The following three offsets are defined in the process table extender :

PDFR - start of the TCB delay chain.

PDEN - end of the TCB delay chain.

PDCNT - number of TCB's currently doing delays in this process.

?TSYS - forward task link offset, defined in the TCB of the task currently doing a delay.

3.4.2 HISLS - The Histogram Active Chain

HISLS - This location, defined in STABLE, contains the PID of the process at the head of the histogram queue.

The histogram queue is a linked list of all the processes which have initiated histogram creation via a ?IHIST call. The histogram queue is a singly linked list off the associated processes process tables. The link offset is "PHLNK" and the link is terminated with a minus one. The information contained in the histogram is stored in the process table extender of the process making the "?IHIST" call. Note that because of this the process making the call must be resident to insure that the process table extender is always in memory.

PHLNK - forward link of the histogram queue, defined in the process table.

The following are offsets defined in the process table extender :

PHPTB - histogram target process table address.

PHWDS - histogram word grouping.

PHST - starting offset of area for histogram information.

PHEND - last offset in area for histogram information.

PHADR - starting address of histogram array.

PHASZ - histogram array map size.

PHTCB - histogram task control block.

3.4.3 CMQWD - The Core Manager Request Queue

CMQWD - This location is defined as a control block offset that only has meaning for the Core Manager control block, CMTSK. It is the beginning of the chain of processes that are enqueued to the Core Manager for swapping, either to be swapped in or swapped out. This offset contains a PID. The queue is a singly linked list, through offset PCMLK of the process table.

3.5 Data Resources

We will also limit our description of data resources to a few items, the largest data resource in AOS being the File System, subject of chapter 5. The data resources we chose to develop here are the Virtual Processors pool, the System Stacks and the Overlays.

3.5.1 The Virtual Processors Pool

Rather than make a call to allocate memory for a virtual processor each time the scheduler wants to bind a process to a virtual processor, a pool of pre-allocated virtual processors is maintained. Then allocating a virtual processor simply becomes picking a free one from this pool. SINIT calls GSMEM during initialization and sets up a chain of virtual processors, i.e. of "resident process tables".

Three entries are defined in STABLE to implement the virtual processors pool:

VPCNT - Virtual processor count
this is the number of virtual processors in the system, both free and in use. Note this could be made a potentially critical tuning parameter. The initial idea was to make VPCNT a simple linear function of the memory size, but VPCNT was actually implemented - in Rev. 2 - as a constant equal to eight, and has not changed since. Clearly performance measurement and evaluation could be done on this value. The scheduler could very well alter this value dynamically, based on load characteristics or other environmental conditions.

VPFCN - Virtual processor free count
this is the number of free (unbound) virtual processors, i.e. a count of the number of entries on VPCHN. Strictly speaking, it is not necessary, but this number may be useful as input to policy algorithms, and certainly will be useful for performance monitoring.

VPCHN - The virtual processor chain
this is a pointer to the first virtual processor in the chain of free virtual processors. This chain is a singly linked list, with offset PLNK in each pointing to the next free virtual processor. A null pointer (-1) in VPCHN denotes the chain is empty, and is also used to terminate the chain.

Note that as a consistency check, VPCHN = -1 if and only if VPFCN = 0.

3.5.1.1 SWPVP - The Swapping Virtual Processor

SWPVP is a virtual processor that the Core manager can use to bind the virtual process table of a process being swapped in or out. One implication of virtual process tables is that assigning a resident process table to a process is independent of allocating/deallocating memory to the process and swapping it in or out. This means the Core Manager must deal with virtual process tables. A difficulty arises from the manner in which swap I/O is performed that makes it necessary to have a resident process table for a process being swapped. The following explanation should make this clear :

For efficiency reasons, swap I/O is done using the system primitive NQBHR - Enqueue Buffer Header (see chapter 5). This allows the entire process image to be read or written in a single disk request. However, one of the offsets in the buffer header that is enqueued, BQMAP, must contain either the physical page number the I/O is to, or the process table address of the process the request is for. In the case of swaps, BQMAP must be a process table address, so that the data channel mapping code in SWAMP can access the process map and perform the read/write to/from the correct memory locations. BQMAP is not a problem for other types of I/O, because in all cases except swap I/O the process will have to be bound to a virtual processor, and thus a process table address can be used.

SWPVP is then a dummy virtual processor that can be used to fulfill the requirements of BQMAP. The Core manager will bind the swapping process to SWPVP before calling SWAPI or SWAPO to swap the process, and will unbind the process after the swap is complete. SWPVP is defined in STKS, as is the Core manager stack CMSTK.

3.5.2 The stacks

In AOS there are eight stacks used by the system :

STACK BASE	STACK DEFINITION
SS	interrupt stack
STK1	per processor stack
CMSTK	core manager stack
ST000	control block 0 stack
ST001	control block 1 stack
ST002	control block 2 stack
ST003	control block 3 stack
ST004	control block 4 stack

At any point in time one of these eight stacks is current, and is defined by words 40-43 in system page zero. The system always uses the B map, therefore all these stacks are defined for the B map. The user is on the A map, therefore switching to the user also changes to his stack. Since the interrupt stack is used with interrupts off for part of the time and the map on part of the time, the logical and physical page for the stack definitions, that is page zero, must be identical. The same is also true for which ever page contains the interrupt stack itself.

LOCATION	SYMBOL	DEFINITION
40	SP	stack pointer
41	FP,CSP	frame pointer
42	CSL	stack limit
43	CSO	stack overflow routine address

As control passes from one routine to another, these locations are continually being re-initialized to point to the current stack information.

3.5.2.1 Interrupt stack

Location INTLV in SZERO is incremented at the start of interrupt handling, and is decremented by the interrupt dismissal code. Thus, we can easily tell when the system is at the base level : INTLV = -1. Only then does the interrupt world execute a VCT instruction with the stack change bit set. If additional interrupts come in while this first interrupt is still being processed, the VCT instruction does not have the stack change bit set.

If the stack change bit is set in the VCT instruction the following occurs that does not occur otherwise :

a) SP, CSP, CSL, and CSO are saved
b) Contents of location 4 is placed in SP
c) Contents of location 6 is placed in CSL
d) Contents of location 7 is placed in CSO
e) CSP is undefined
f) The old SP, CSP, CSL and CSO are pushed.

When interrupt processing at the base level completes, the previous stack is restored using the RSTR instruction, except in the special case where we would return to the checksum loop in the scheduler, and there are active control blocks; under these circumstances restoring the stack is not necessary, because the stack is not used until it is reinitialized as the per processor stack.

Interrupt stack definitions :

```
stack base: SS
stack limit: SSLMT
stack size:  255. words
stack is defined in module STKS
loc 4: SS, interrupt stack base
loc 6: SSLMT, interrupt stack limit
loc 7: OVFLO, stack overflow processing entry point (in INTS)
INTLV: interrupt level indicator, set to -1 at base level
```

SINIT, the system initialization code, assumes that STKS is the first module, and SS the first location within the module, that cannot be write protected. All system pages between page zero and SS are write protected as invariant system code.

3.5.2.2 Per processor stack

The per processor stack acts as the universal stack for all system activity not related to one of the other seven system stacks. This stack being constantly re-initialized by code paths in the Scheduler and the Core Manager, it is necessary that all code using this stack be very careful that it will not depend on it in cases where it may be changed.

Per processor stack definitions :

```
stack base: STK1
stack limit: STK1N (stored at location STK1-1)
stack size: 100. words
stack is defined in module STKS
PRSTK: STK1  is a label defined in SZERO
```

3.5.2.3 Core manager stack

For some things the Core manager requires a stack which will not be changed asynchronously by another code path, like the Per processor stack is. Therefore the Core manager has a stack of its own as well as a control block which resides as the first entry on the eligible queue.

Core manager stack definitions :

```
stack base: CMSTK
stack limit: CMSEN (stored at location CMSTK-1)
stack size: 200. words
stack is defined in module STKS
.CMSTK: CMSTK  is a label defined in STABLE
CMTSK: Core manager control block, defined in STABLE
```

The stack limit and control block pointer are defined in the same manner as control block stack information, therefore CMSEN is located at (CMSTK-1), and CMTSK at location (CMSTK-2) points to CMTSK.

3.5.2.4 Control block stacks

As mentioned before, each control block has associated with it a stack. For the special control block used by the Core Manager, the stack is called CMTSK. This section will discuss the other control blocks and there related stacks.

There are five (5) control block stacks. The first one (ST000) is defined at assembly time in STKS and has its base address pre-bound in STABLE. The other four control block stacks are defined at system initialization time by SSOV1.

All the stack definitions follow the same format. Let N be a number from 0 to 4 representing one of the five respective control block stacks. We have :

The stack base is defined as STN00. The stack limit is defined at the stack base minus one (location STN00-1) and the pointer to the corresponding control block CBN00 is stored at the stack base minus two (location STN00-2).

Because ST100 through ST400 are defined at system initialization time, whenever these stacks are referenced by any system code path, the address of the desired stack is loaded as an offset from PS000. PS000 is defined at bind time to contain the stack base address of ST000, and the four offsets following PS000 in STABLE are filled at system initialization to point to the other four system stacks. At runtime we then find in STABLE :

```
PS000:      ST000          Note : bit 0 is set in
PS000+1:    ST100          ----   the appropriate
PS000+2:    ST200                 PS000+X when the
PS000+3:    ST300                 corresponding
PS000+4:    ST400                 stack is in use.
PS000+5:    -1 (terminator)
```

The control block CB000 for stack ST000 is defined immediately after the stack, i.e. the stack limit is one less than the beginning of the control block. CB100 to CB400 are not defined to immediately follow ST100 to ST400 respectively. Each stack is 256. words long, and each control block is 21. words long.

When a control block is scheduled (in SCHED), its stack is initialized from the parameters given above if this is the initial run for the control block. If the control block was already running, the previous SP and CSP (FP) are stored in stack offsets CSTKC and CSTK of the control block.

If control block was not running, locations 40-42 are set up as follows :

SP: stack name (STN00)
FP: stack name -1 (STN00-1)
CSL: contents of (STN00-1)

If control block was active, locations 40-42 are set up again as follows :

SP: offset CSTKC
FP: offset CSTK
CSL: contents of (STN00-1)

3.5.3 Overlays

Overlay segments contain system code that is usually not permanently resident but is brought into memory at some point in time. In AOS we distinguish between two categories of overlays , so called "disk based" overlays and "resident" overlays. Disk based overlays are read in only when needed, and kept in according to certain factors like their use count and time of last reference. Resident overlays are copied to core at system initialization time and will never have to be read in again.

The system reserves 1Kw of its address space for holding overlays, page 30. (logical addresses 74000-75777). This 1Kw area is divided into two 1/2 Kw pieces, each of which may contain a single overlay : this restricts overlay segments to a length of 512. words, or 1000 octal. Thus an overlay, when in system space, will begin at address 74000 or 75000. More than two overlays may be resident in memory at any time, however only two can ever actually be mapped into the system's address space. If an overlay is referenced that is currently in core, the system need only map the physical page containing the overlay into the 74000 address slot.

All overlay segments are contained in the system file, where they are in order by overlay number, starting on the first 2000 boundary after location CLIEN (also referred to as ?NMAX). At system initialization time SINIT will store into OVFAL and OVFAH the disk address of the beginning of the overlay area allocated by SYSBOOT. Each overlay segment being 512. words long (1000 octal), hence occupies two disk blocks. This fact is used by either SINIT or SOVLY when reading overlays in, the block number of the beginning of the overlay within the area being simply two times the overlay number.

The only I/O operations to the overlay area are read operations. Overlays are pure code; they are never modified and thus never need to be written back to the overlay area.

3.5.3.1 Resident overlays

Resident overlays can be regarded as virtual system code paths; they are readily accessible by simple remapping operations. For some things like processing direct type system calls (see chapter 6), or driving data channel devices, AOS could not afford to wait for a possible read in of an overlay from disk. Therefore a fixed - and ever growing - number of system overlays are being defined as resident and read in at system initialization onto physical pages of memory which will never be re-allocated.

The control table for resident overlays, ROVTB in STABLE, is filled by SINIT to indicate where each overlay resides in memory. Its length is defined by SCNOR in PARS. There is one entry per resident overlay, with some interspersed null words to keep overlay numbers in order. The code in module SCPRC marks overlays for permanent residency by defining accumulating symbols which are resolved at bind time and will be used by SINIT to build ROVTB.

A list of resident overlays which will always be loaded is given in table 3.1. In addition, other resident overlays can be present, depending on which devices are sysgen'ed. There will be at most 33. resident overlays in any given system.

Note that the resident overlay numbers in the resident overlay symbols are different - one less than - the overlay numbers as found in the system file (table 2.2.6). This is because overlays #0 and #1 need to be core resident only for system initialization, hence are not included in ROVTB. This also means that offset zero in ROVTB is the entry for overlay #2.

Symbol	ROVTB offset	Overlay name
RO001	0	CMOV1
RO002	1	SCMOD
RO003	2	IOOV
RO004	3	SWAPI
RO005	4	SWAPO
RO006	5	SWAPM
RO007	6	ECC
RO010	7	DVRS
RO011	10	DVRS1
RO017	16	RROV1
RO020	17	RROV2
RO027	26	DVOV6
RO030	27	RROV3
RO032	31	VDUMP
RO035	34	RRESO
RO037	36	DZOV2
RO041	40	IOOV1
RO042	41	IRECO
RO043	42	DRSND

Table 3.1 - Resident Overlays

Entries in ROVTB are used as map words for mapping the overlay to slot 74000. They are ordered by overlay number and have the following format :

- entry N = XB0 + 74000 + phys. page # of page holding overlay #(N+2)

  where X=0 if the overlay resides in the first half of the page
  or X=1 " " " " " " second " " " "

- entry N = 0 means overlay #(N+2) is not a resident type overlay, or is not needed by this particular system.

Immediately following ROVTB a dummy header RDHDR is defined, in the same format as for regular overlay headers (see section 3.5.3.6). It is filled by the resident overlay calling code in SCPRC every time a resident overlay is about to run, and location 366 (CRSEG) will point to it with bit zero set (1b0) when the overlay is effectively running.

3.5.3.2 Resident overlay calls

To call an entry point named 'ROO' in some resident overlay segment, the programmer writes code equivalent to the following :

```
        JSR@    .ROVC
        ROO             ;ROO must be declared .EXTN
        .
        .
        .
.ROVC:  ROVCL           ;ROVCL must be declared .EXTN
```

That is, the programmer actually does a call to the entry point ROVCL (which exists in module SCPRC) and specifies the actual entry in the overlay by placing it in line immediately after the call to ROVCL. The AC's are preserved across this call and must be loaded with whatever values the entry expects; the return from the entry will be to either the second or third word after the call to ROVCL, depending on whether the error return is taken, or whether there even is an error return defined by the overlay entry.

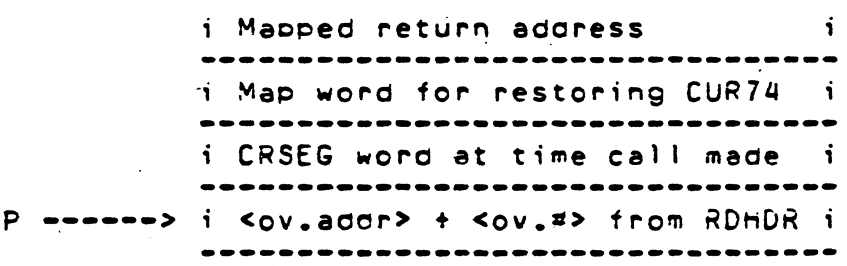
The binder will resolve the external reference to 'ROO'; the actual value filled in for 'ROO' will be of the form

<180 + overlay #> (left byte) + <offset of entry> (right byte)

For example, entry 'GCORE' is at offset 174 within overlay RROV1, overlay # 20. Any reference to GCORE will then resolve as

180 + 20*400 + 174 = 110174

The ROVCL mechanism works via a virtual return block left on the stack. This is invisible to the caller and callee except for the space it takes up. The format of this virtual return block is :



ROVCL can also be entered (to ROVC1) by passing the argument in AC0. This is used for example by RRESD to restart a device. This code being device independent, in line passing of the argument cannot be used as the restart routines are different for each device. Therefore the restart routine address is loaded in AC0 from the appropriate device DCT (offset DCPRS). The format of the entry in DCPRS is the same as for an in line argument, i.e. <180 + overlay #> + <offset of entry in overlay> .

3.5.3.3 Disk based overlays

A parameter defined in PARS, SCOVM, sets the minimum number of disk based overlays to be kept in core at any time by the system. This is to reduce the number of times an overlay is actually read from disk.

The following databases are maintained by the system in order to implement disk based overlays :

- The overlay table, OVTAB, which gives the current status of each overlay segment.
- The OTIME table, which gives the time of last reference for each overlay segment.
- The overlay chain, OVMCH, a least recently used list of headers describing overlay segments currently in core.
- The following variables, used as listed. Unless otherwise noted, all of these are defined in the module STABLE :

  OVCNT - the # of items on OVMCH, the overlay chain

  OVFCT - the # of overlay segments on OVMCH that are free currently, i.e. do not have paths executing in them

  OTMIN - the minimum length of time that an overlay will be kept in core (defined in module SZERO)

  OVYEN - a pointer to the last entry on OVMCH

  OVFAH & OVFAL - double precision logical disk address of the overlay area

  OVCLH & OVCLL - double precision count of OVLAY and OVCHN calls

  OVRSH & OVRSL - double precision number of times any overlay is in core when called

  OVWT - a flag set when waiting for memory for overlays

- Absolute location 366 (CRSEG), in system's page zero, which points to the OVTAB entry of the currently active overlay segment.

3.5.3.4 Disk based overlay calls

Since the system must determine at the time the call is made whether the referenced overlay segment is currently in core, overlay calls are not made directly. Instead, all calls to disk based overlays are made through the module SOVLY, by calling one of the entries in SOVLY and specifying the actual overlay entry to be called.

There are essentially two ways to call an entry in an overlay segment : a straight call that returns to the caller just as a normal subroutine, and a chain call that returns not to the caller, but to the caller's caller. There is also a special call used when the target entry is contained in the same overlay segment as the call is made from. The use of each of these is explained below.

OVLAY calls

To call an entry point named 'FOO' in some overlay segment, the programmer writes code equivalent to the following :

```
        JSR@    .OVLY
        FOO             ;FOO must be declared .EXTN
        .
        .
.OVLY:  OVLAY           ;OVLAY must be declared .EXTN
```

That is, the programmer actually does a call to the entry point OVLAY (which exists in the module SOVLY) and specifies the actual entry in the overlay by placing it immediately after the call to OVLAY. The AC's should be loaded with whatever values the entry expects; the return from the entry will be to either the second or third word after the call to OVLAY (i.e. the first or second word following 'FOO' in the above example), depending on whether the error return is taken, or whether there even is an error return defined by the overlay entry.

The binder will resolve the external reference to 'FOO'; the actual value filled in for 'FOO' will be of the form :

<160 + overlay #> (left byte) + <offset of entry> (right byte)

For example, if 'FOO' is at offset 126 (octal) within overlay segment 104 (octal), then FOO gets replaced by

(1 + 104*400) + (126) = 142126

This implies overlay numbers range from 0 to 177 and that entry offsets range from 0 to 377 .

OICAL calls

When making an OVLAY call to an overlay entry that is in the segment making the call, the entry OICAL is used. Its use is analogous to OVLAY, and it functions in the same manner. Indeed, OVLAY could be used instead, but OICAL takes advantage of the fact the call is from the same overlay as the target entry to greatly simplify processing the call. Thus it is much more efficient to use OICAL instead of OVLAY in making such a call. Note in this particular case, the overlay entry is not declared external, and the reference gets resolved simply to the offset within the overlay.

OVCHN calls

Overlay chain calls are made in a manner analogous to OVLAY calls, but the call is made to OVCHN instead of OVLAY. The crucial difference is that control will never return to the place of the call. Instead, when the called entry returns, it will return to the caller of the routine making the OVCHN call, as in the following example:

```
        Module FOO                  Overlay BAR                 Overlay FOOBAR
            .                           .                           .
            .                           .                           .
        JSR  @.OVLY         BAR1:       .                 FBAR1:    .
        BAR1                            .                           .
            .                       JSR  @.OVCN                     .
            .                       FBAR1                       RTN
 .OVLY: OVLAY                           .
                                        .
                                .OVCN: OVCHN
```

Assume module FOO (which may or may not be an overlay itself) makes an OVLAY call to entry BAR1 in overlay BAR. Since this is an OVLAY call, FOO expects to regain control. Entry BAR1, however, makes an overlay chain call to entry FBAR1 in overlay FOOBAR. Since this is an overlay chain call, BAR will not be returned to when the return statement in FOOBAR is executed. Instead, control returns to the caller of the OVCHN caller, namely FOO.

An overlay chain call then is really nothing more than a special JMP instruction, i.e. JMP between overlays. It is typically used when not all the processing necessary for a call can be done in a single overlay segment due to the 1/2 Kw size limit on overlays. Thus, if the code for a routine to be made an overlay is longer than 1/2 Kw, it can be split into two or more overlays. When the processing in the first part is complete, control is transferred to the second part by making an OVCHN call.

PROC, PROC2 and PROC3 are examples of such chaining. These three overlays contain the code that implements process creation, the first part of which is done in PROC, which then makes an OVCHN call to PROC2, which in turn chains to PROC3. Upon completion of PROC3, control returns to the point of the original PROC call.

3.5.3.5 The overlay tables OVTAB and OTIME

The current status of each overlay segment is maintained in a table, OVTAB, defined in STABLE. The entry at offset N from the beginning of this table describes the status of overlay N. There are three possible values an OVTAB entry may contain :

- 0 - means the associated overlay is not currently in core, or is a resident type overlay having its entry in ROVTB
- 1 - means the overlay is not currently in core but is being read in
- >1 - any other value indicates the overlay is in core; the value is a pointer to an eight word header describing the usage of the overlay segment. It is these eight word headers that are linked together to form the overlay chain.

The length of OVTAB is defined by the system constant SCNSO, number of system overlays (in PARS). OVTAB is maintained by SOVLY.

There is a second table defined in STABLE that records for each overlay segment the time of its last use. This table's starting location is defined by the symbol OTIME; the length of the table is determined by the value of SCNSO and is equal to the length of OVTAB.

Offset N from OTIME is the time of day, in seconds, that overlay N was last called. (Actually, it is the low order word of the time of day, TODL in SZERO.) This table is used when searching for an overlay to remove from memory; when an overlay has been found that is a candidate for removal, its time of last reference is compared to the current time. If the difference is less than OTMIN, the overlay is not removed.

A zero value at offset N in OTIME means that the overlay number N either has not yet been called; or is a resident type overlay for which keeping track of its time of last reference would be irrelevant, this type of overlay never becoming a candidate for removal from memory.

3.5.3.6 The overlay memory chain OVMCH

Whenever an overlay is in core, an eight word memory header is associated with it. All such headers are chained onto a doubly linked list called the overlay memory chain. OVMCH points to the first header on the chain, OVYEN to the last.

This list is maintained in least recently used (LRU) order. OVYEN therefore points at the header describing the last overlay that had (or has) a path executing in. This is done so that when the system needs memory and decides to remove overlays from core, those overlays not referenced for the longest time are removed. OVMCH thus points at the least recently used overlay's header. OVCNT is a count of the number of headers linked together (and OVCNT/2 the number of 1Kw pages the system has allocated for overlays). A system constant, SCOVM, defines the minimum value of OVCNT. Although the number of overlays kept in core at any time may increase as long as memory is available and not needed for other purposes, however memory used for overlays will only be reclaimed until this minimum value is reached (i.e. until the minimum number of pages of memory devoted to overlays, SCOVM/2, is reached).

The offsets in the headers are described in full below. See PARS for the actual definition.

CMSFL - pointer to next header on chain (terminated by -1).

CMSBL - pointer to previous header on chain (terminated by pointing back to the head of chain OVMCH).

CMOVB - the physical page # of the 1Kw page of memory that actually contains the overlay. This is needed for releasing the memory, and also for mapping the overlay into the system's overlay window.

CMOV1 - the overlay # of the overlay this header describes, or -1. If -1, there is no overlay currently resident in the page of memory this header describes.

CMAD1 - the base address of the overlay area for the overlay described by this header.This will be either 74000 or 75000 (or 0 if CMOV1 = -1). This gives the address in the system's address space that the overlay will be mapped into.

CMAD2 - the address of this header's buddy header. Since two overlay segments will fit into 1Kw of core, each time a 1Kw page of memory is allocated for use by overlays (or released), two headers are needed which describe the overlays that the page can contain. Therefore a 16. word area of GSMEM space is requested for building the overlay memory headers and divided into two eight word parts which are linked to one another by this offset. This has to be so, because when the area used for headers is given back to the system, both 8 word headers that came from a 16. word area must be returned.

CMOUC - the current usage count. This is the number of calls that are currently active in this overlay. Evidently, if this number is greater than 0 the overlay cannot be removed from memory.

The last word in the overlay memory header, offset seven, is not used.

3.5.3.7 The SOVLY module

This section describes in some detail the workings of the SOVLY module. The approach taken here is to follow a sample overlay call from the time the call is made, through SOVLY until control is actually passed to the overlay entry point; and again from the time the overlay entry returns through SOVLY back to the point of the original call. Of particular interest is the way the stack is handled during this sequence of steps.

OVLAY calls

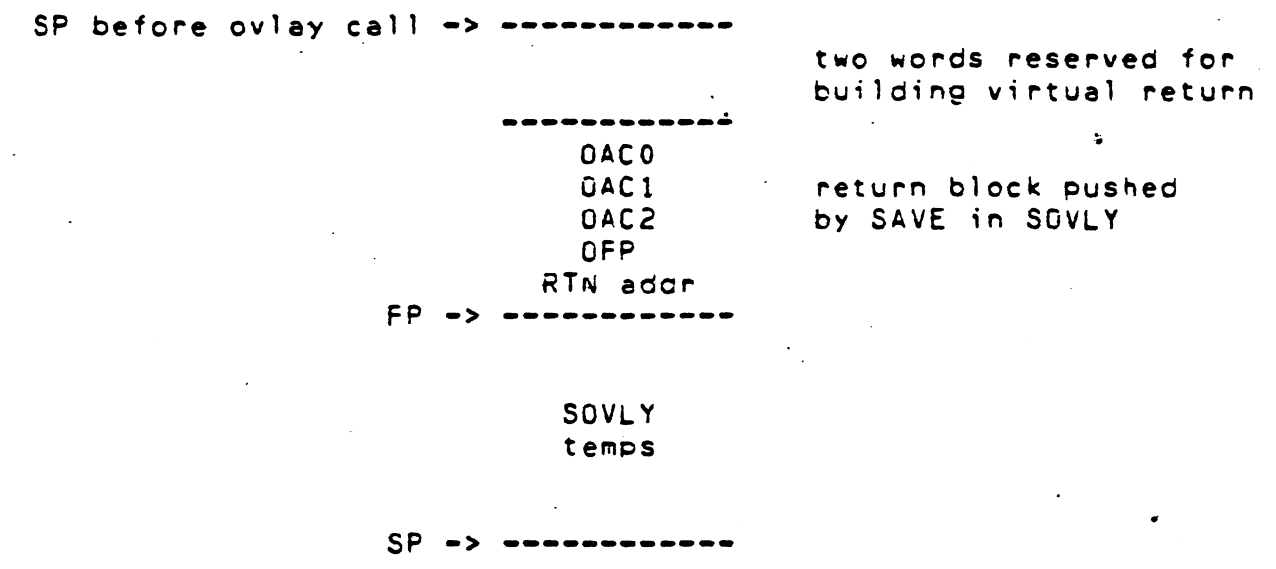
Assume the following overlay call is made :

```
EJSR    OVLAY
IGRNT
```

(IGRNT is an entry in overlay segment SSOV4 that returns various runtime statistics on a process.) We assume the AC's have been properly loaded.

The first thing SOVLY does, even before doing a SAVE, is to increment the stack pointer by two. The two stack words so reserved are used to hold a virtual return address, the first word of which is the overlay number of the calling code (or -1 if the call is from resident system code). The second word of the virtual return address is the offset of the return address within the overlay (or offset within the system resident code). A SAVE is then done, building a stack frame for use by SOVLY and saving the caller's AC's.

At this point the stack looks like the following:



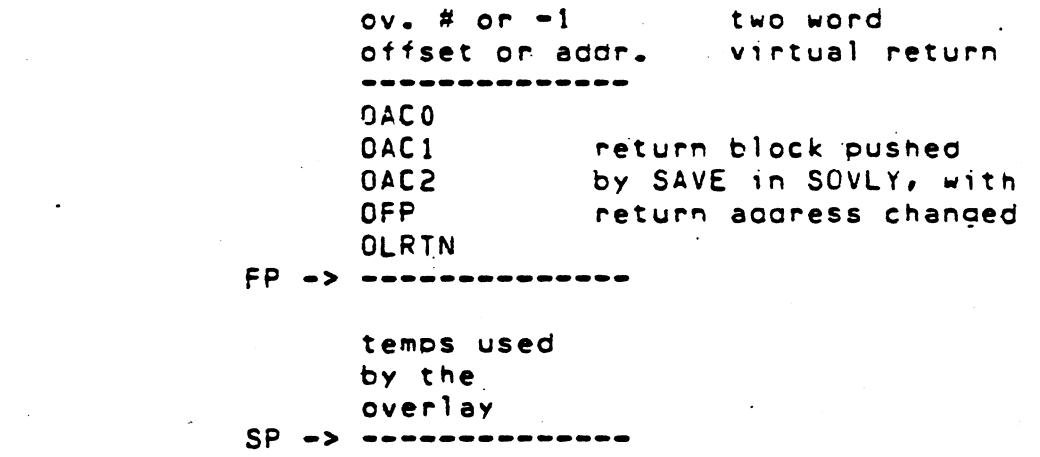
The return address is ISZ'ed to point to the actual return location rather than the word containing the overlay entry (IGRNT in our example). SOVLY now checks to see if an overlay is currently active by looking at CRSEG (i.e. if the call is from an overlay). If so, the return address (offset within the overlay) is stored in word two of the virtual return and the overlay number stored in word one. If the call is from resident code, -1 is stored for the overlay number. The return address in this case is an absolute address within the resident code.

Next, the overlay entry word is broken into the number of the overlay to be called and the offset within that overlay segment to which control is to be transferred. The overlay number is used as an index into OVTAB to determine the current status of the overlay. At this point there are three cases to consider, depending on whether the overlay is in core, being read in, or not in memory currently.

a) The overlay is in core

This is the simplest case. CRSEG is set to point to the OVTAB entry for the overlay, indicating this is the currently active overlay. Using the physical block number in the header, the actual 1kw page that contains the overlay is mapped into the system's 74000 address slot with the write protected bit set. The overlay header is moved to the end (most recently used position) of the overlay chain, and the OTIME table is updated to make the time of last reference the current time. The base address of the overlay (also obtained from the header) is added to the overlay offset previously determined; the result is the actual address control is to be passed to. The location immediately before this full overlay entry address is examined, as by convention it must hold the number of stack temporaries the entry will use. The current stack pointer is incremented by the number of stack temporaries thus creating the stack frame for the overlay entry. The return address is changed to equal 'OLRTN', an entry in SOVLY that handles returns from overlays (see the next section). The AC's are now re-loaded with the values at the time of the call to SOVLY, the full (mapped) address of the overlay entry is pushed and a POPJ done which passes control to the overlay entry.

At this point, control is now in the called overlay and the stack looks like this :



Note the overlay entry can reference the AC's passed to it just as in any other subroutine call, and in particular can store into them to pass values back. The return address however, points to OLRTN in SOVLY, thus when the overlay does a 'RTN' SOVLY regains control.

b) The overlay is being loaded

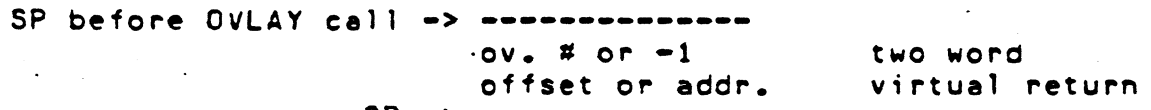
In this case, the process making the overlay call is penoed. When the process is unpended, processing of the call continues by looking again in OVTAB to determine the current status of the desired overlay. This time, OVTAB should indicate the overlay is in core, and processing continues exactly as in case a.

c) The overlay is not in core

If the overlay is not in core and not being loaded, it must be read in. First, though, memory must be allocated to hold the overlay. To do this, the chain of overlay headers is examined, starting with the least recently used element - pointed to by OVMCH -. The initial entry may be empty, in which case it is immediately grabbed. More commonly, the entry will be in use, i.e. describe a piece of memory holding some other overlay. In this case, the time since the last reference to the overlay described by this header is computed from the current time and the last reference time in the OTIME table. If greater than OTMIN, the use count is checked, and if the use count is zero the header is chosen. OVTAB is changed to indicate the overlay currently described by the header is no longer resident. An NQBHR call is made (see chapter 5) to read in the overlay from the overlay area on disk, using an area on the stack as the buffer header for the call and the memory area described by the overlay header, that is 1000 words starting at 74000 or 75000, as the buffer. When the read completes, processing continues as for case a.

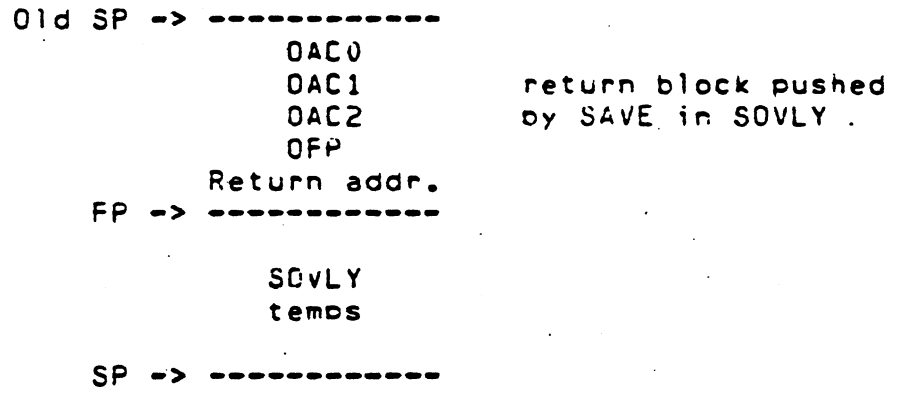
If a suitable overlay header cannot be found on OVMCH (i.e. there are none with a zero use count and not referenced recently), an attempt is made to expand the amount of memory used for overlays. A call to GSMNW is made to get a 16. word free area, followed by a call to GMBLK to get a 1Kw page to be used for holding two overlays. If both calls are successful, the 16. word free area is divided into two 8 word overlay headers describing the 1Kw page; these two headers are put on OVMCH, incrementing OVCNT and OVFCN appropriately. If either allocation fails the process is pended, using a special key denoting the process is waiting for an overlay area, after waking up the Core manager and signalling that memory is needed for overlays. After the process is unpended, more free memory should be available and the search for a free descriptor on OVMCH starts again.

When the overlay code completes, it executes a RTN instruction that passes control back to the SOVLY module at either OLRTN or OLRTN+1 depending on whether the return address was ISZ'ed by the overlay, i.e whether the error return was taken or not. When SOVLY regains control, the contents of the AC's are as they should be restored to the caller, and the stack is as follows:



If the OLRTN+1 return is taken, the offset in the virtual return is ISZ'ed so that it points to the caller's good return. The routine OVREL is called to release the overlay segment just returned from by decrementing the use count in the overlay header. If the count becomes zero, the count of free areas (OVFCT) is ISZ'ed, and any processes pended waiting for an overlay area are unpended.

Now the return to the overlay caller can be made. The overlay number and the offset are popped off the stack and stored in PTMP1 and PTMP2 respectively (these are locations in SZERO used by the system to store temporaries). A SAVE is then executed, which re-establishes a stack frame for SOVLY and saves the modified AC's returned by the overlay that must be passed back to the caller. The stack now looks like this:



If the call was not from an overlay (i.e. PTMP1, the virtual return overlay number, equals -1), a simple return to the caller is now done by storing PTMP2, the return offset, in the return location of the return block pushed by SOVLY and by executing a RTN instruction. If the call was from an overlay however, the return must be done in much the same manner an OVLAY call is done. A check is made to see if the overlay being returned to is resident, etc. in exactly the same way as previously described - we actually use the same code path -. Once the overlay to be returned to is found, it is made the current overlay segment and a return can be done to it by storing the mapped return address in SOVLY's return block and executing a RTN instruction.

OVCHN calls

The steps performed when making an overlay chain call (via OVCHN) are almost identical to those described above for an overlay call via OVLAY. The chief difference is the handling of the stack. Recall that a chain call does not return to its caller, hence no virtual return is built on the stack and the overlay does not return through SOVLY.

Assume the stack prior to the OVCHN call is:

```
FP before OVCHN call -> ------------
(OFP in what follows)                    stack frame used
                                         by caller of OVCHN
                                         (not returned to)
SP before OVCHN call -> ------------
(OSP in what follows)
```

The initial processing in SOVLY is similar to that for an OVLAY call. Since no virtual return is built, the stack while in SOVLY looks slightly different:

```
                OFP -> ------------
                                         stack frame used
                                         by caller of OVCHN
                                         (not returned to)
                OSP -> ------------
                            OAC0
                            OAC1         return block
                            OAC2         pushed by SOVLY
                            OFP
                       entry addr.       full entry address
                 FP -> ------------

                          SOVLY
                          temps

                 SP -> ------------
```

however, the final processing before passing control to the overlay entry is quite a bit different as the stack must be handled in another way. The stack pointer (SP) is set to equal the current frame pointer (FP). The number of stack temporaries used by the overlay entry is added to the old frame pointer to give the new stack pointer, which is saved for now in AC0. The frame pointer is now set to the old frame pointer value, obtained from the return block. Hence the stack now looks like this :

```
          FP -> ------------
                                stack frame used
                                by caller of OVCHN
                                (not returned to)
         OSP -> ------------
                    OAC0
                    OAC1        return block
                    OAC2        pushed by SOVLY
                    OFP
                  entry addr    full entry address
          SP -> ------------
```

The stack pointer now points just after the mapped overlay entry address which has been computed and stored in the return address location of the return block pushed by SOVLY. This address is popped and saved in PTMP1. OFP, OAC2, and OAC1 are popped into AC's 3 to 1 respectively. This sets up the AC's for the called entry, except for AC0, which still has the new stack pointer. OAC0 is now popped into AC3, the new stack pointer in AC0 stored in location 40 to make it the current stack pointer, and AC3 (really OAC0) moved to AC0. Control is then passed to the overlay by executing a JMP @ PTMP1. The stack is now:

```
          FP -> ------------
                                stack frame used
                                by caller of OVCHN
                            (now used by the overlay)
          SP -> ------------
```

Note the overlay will use the same stack frame as its caller. If both use the same number of stack temporaries, they can thus share the same frame, and stack variables set up by the caller can be used by the called overlay without having to be recomputed or passed in the AC's. Also, no virtual return has been pushed on the stack, thus the overlay does indeed return directly to the caller of its caller. (Which could, of course, be an overlay, causing the return to go back to SOVLY after all).

OICAL calls

A call via OICAL is handled like a call to OVLAY. In fact, the same code could be used in both cases, and calls made using OICAL code be made using OVLAY instead. However, OICAL can handle calls to overlay entries residing in the same segment as the call much more efficiently because it takes advantage of the fact that the target overlay is known to be resident and already mapped to the system's overlay window. Hence all that needs to be done is build the virtual return on the stack, set the return address so that the return from the overlay comes back to SOVLY, and give control to the overlay entry.

3.5.3.8 General considerations

- SOVLY interacts with the Core manager in two ways. The first of these has already been mentioned - namely, if no memory is available when SOVLY tries to grow the overlay pool, the Core manager is wakened and signalled that more memory is needed for overlays. The second interaction between the Core manager and SOVLY is the deallocation of overlay memory. SOVLY never deallocates memory it has obtained for overlays; instead this is done by the Core manager. The Core manager, when it runs, checks to see if OVCNT, the number of items on the overlay chain, is greater than SCOVM. If so, the Core manager tries to reclaim some of the memory allocated to overlays. This is done by searching OVMCH for a header describing a free or not recently referenced overlay segment. If one is found, its buddy header must also be checked to see that the overlay it describes can also be removed from memory. (Again this is necessary because each 1Kw page holds two overlay segments, and thus is described by two overlay headers.) Assuming this is the case, the page may be claimed, once OVTAB is updated to indicate the overlays held in the page are no longer in core. Note the memory used for the headers is also returned at this time. The code which does this deallocation is in the module COREM; it is straightforward and will not be discussed in further detail.

- The code at an entry point in an overlay should not do a SAVE, since the stack is already set up by SOVLY when the overlay entry point is reached. AC3 will contain the frame pointer. By convention, the location before the entry point must contain the number of stack temporaries to be reserved for the entry's stack frame (i.e. the number that would be the argument to the SAVE instruction if there were one). AC's 0-2 will be as the caller left them.

For example, in overlay SSOV4 we find :

```
          .
          .EXTN   IGRNT
          .
          .

          6                       ;# of stack temps
IGRNT:    EJMP    GRNTI           ;real code is at
          .                       ;end of overlay
          .
```

Note this is different for resident type overlays. In that case RGVCL does not do the SAVE, hence the first instruction at the overlay entry point should be a SAVE N, where N is the number of temporaries needed by this particular entry.

For example, in overlay RROV1 we find :

```
          .
          .EXTN   GCORE
          .
          .
GCORE:    SAVE    2
          .
          .
```

- The code in overlay segments must be both pure and relocatable. That is, it may not modify itself, either by changing one of its instructions or by storing into a variable located within the overlay. All variables used by an overlay must be either stack variables or be externally defined. And overlay segment code must also be relocatable. This means, for example, that a dispatch instruction to transfer control within an overlay cannot be used, since the dispatch table requires absolute addresses.

- Mixed type overlay calls are restricted as follows :

Resident type overlays can be called either from kernel code, another resident type overlay or a disk based overlay. This, because the virtual return block built by ROVCL saves the CUR74 map word as well as the mapped address to return to.

Disk based overlays can be called from either the kernel or another disk based overlay, but not from a resident type overlay. The way the OVLAY routine is presently written, a minus one (-1) is stored as first word of the virtual return if the call is from the kernel or (would be) from a resident type overlay, and the overlay number if the call is from another disk based overlay. The return address is stored as second word of the virtual return block. Assuming this return address would be in a resident overlay, it would be a 74xxx or 75xxx mapped address. But OVLAY does not save the current 74 map word in the virtual return,...hence it cannot work.

One way to do it would be to store CUR74 instead of -1 in case the call was coming from a resident overlay - using the fact that the write protect bit would be set to differentiate this map word from a plain overlay return address -, then change the overlay post-processing routine to check for the presence of this map word and perform a remap for returning to the resident overlay.

CHAPTER 4 - PATHS AND STATES
(updated for AOS Rev. 3.11)

4.1 Introduction

This chapter discusses some of the major system paths found in AOS. These are the heart of the product. They are:

4.2	Scheduler	-- Both the AOS process scheduler and the user task scheduler are discussed in this section
4.3	COREM	-- The AOS memory manager. This routine performs a great deal of scheduler oriented functions.
4.4	Interrupt world	-- How AOS responds to interrupts.
4.5	Misc.	-- Some of the functions that span various modules of AOS. For example: process termination and creation, ?DELAY, daemons, and a description of the AOS timeslices.

4.2 The AOS scheduler

There are three distinct parts to the AOS scheduler. These are:

1. The AOS process scheduler (SMON in SCHED). Its responsibility is to determine if any process tables (PTBL) or controls block (CB) are ready to run, and if so, run them. If not, it will perform a checksum on various AOS constants and shuffle the ELQUE and VELQUE entries to give everybody a chance.

2. The AOS task scheduler (PSCHED in SCHED). Its purpose in life is to decide which user TCB is to run when the process gets control of the CPU.

3. COREM (Core Manager), which in AOS is not actually part of the scheduler, but does a great deal of scheduler oriented activities. COREM handles all memory allocation on a page basis including swapping and GSMEM / GVMEM pages.

4.2.1 Definitions

An ELIGIBLE process is a process that has both its primary and secondary contexts currently resident in memory. The process must also be bound to a virtual processor. (i.e. not blocked). In the terminology used by the AOS designers, a RESIDENT process that is waiting for some event (I/O or otherwise) is still considered ELIGIBLE.

A BLOCKED process is a process of the PREEMPTIBLE or SWAPPABLE type that has been pended for some period of time. A BLOCKED process may be either ELIGIBLE or INELIGIBLE (i.e. he may be currently resident in memory or he may be swapped). The BLOCKED process, therefore, is a process that is a candidate for swapping if the system needs the memory.

- A process becomes blocked as soon as the scheduler finds there are no ready TCBs for that process, or by the explicit ?BLKPR system call.

A INELIGIBLE process is a process whose primary and secondary contexts currently exist only in the SWAP file. An INELIGIBLE process must be first made eligible (i.e. it must be swapped into memory) before it can be bound to a virtual processor.

4.2.2 Queues

AOS maintains 5 queues. The databases on these queues are of the following type:

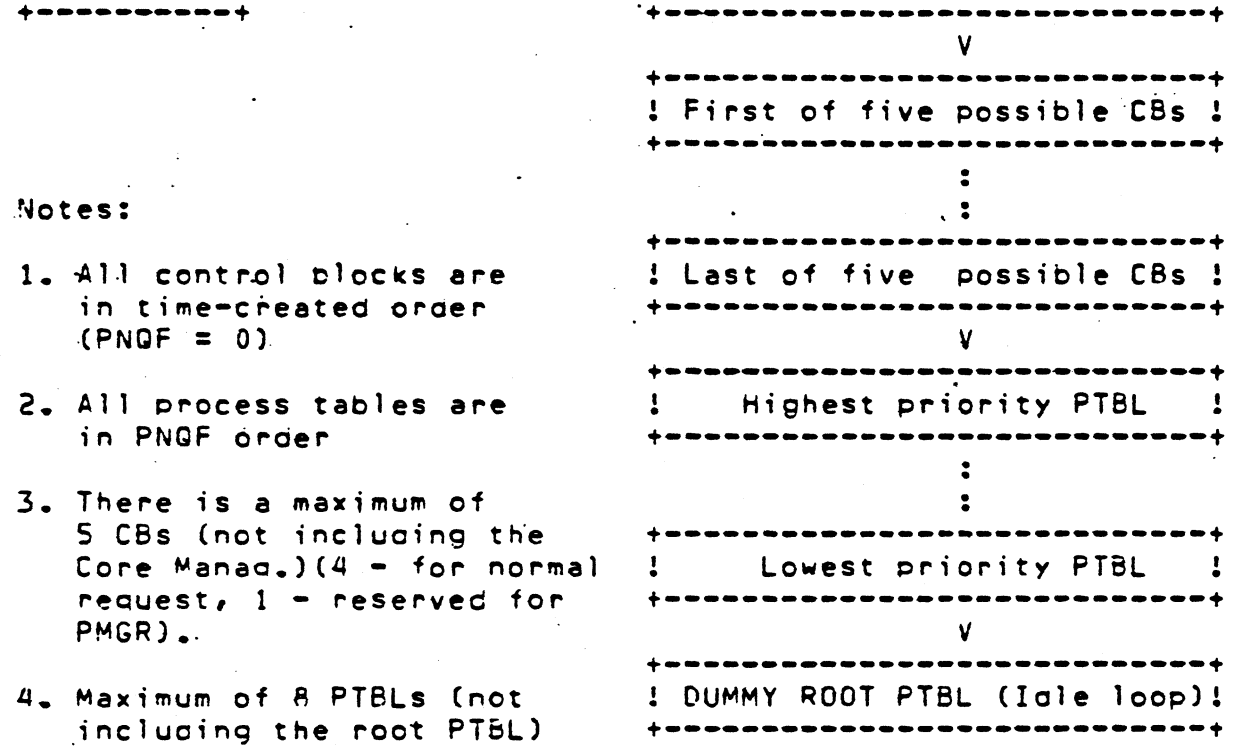
PROCESS TABLES - Contain enough information about a user process to allow the scheduler to both make a decision about scheduling the user, and then give control of the CPU to that user process (see PARS.SR).

CONTROL BLOCKS - Contain the static (non-stack) state of a path within the system. These paths are active to either process user system calls, or daemons created by the system. If the user system call is non-direct, or the daemon request requires a stack, a system stack will contain the dynamic state of this path. The dynamic state of a path includes all subroutine return addresses, and temporary variable data used by the path (see PARS.SR).

The queues and the pointers to the queue head are:

ELGUE: The eligible queue of process tables and control blocks. This is the primary queue used by the scheduler.

* The CORE MANAGER Control Block is always first on this queue.
* The active Control Blocks are next. These are in FIFO order and reflect system daemon requests and user system calls.
* The bound process tables are next. These are in an order derived from a process's priority enqueue factor (PNQF). These include the PMGR process table which is permanently bound to a virtual processor.
* Last on queue is a dummy process table, the root process table. This never requires CPU time, but is used to mark the end of the elgue. The root process table has a PID of 0, and is the father of the PMGR and OP:CLI processes.



Notes:

1. All control blocks are in time-created order (PNQF = 0)

2. All process tables are in PNQF order

3. There is a maximum of 5 CBs (not including the Core Manag.)(4 - for normal request, 1 - reserved for PMGR).

4. Maximum of 8 PTBLs (not including the root PTBL) (The PMGR is always on ELQUE).

Other queues:

VELQUE: Queue of unbound eligible processes (a virtual queue also in in PNQF order)

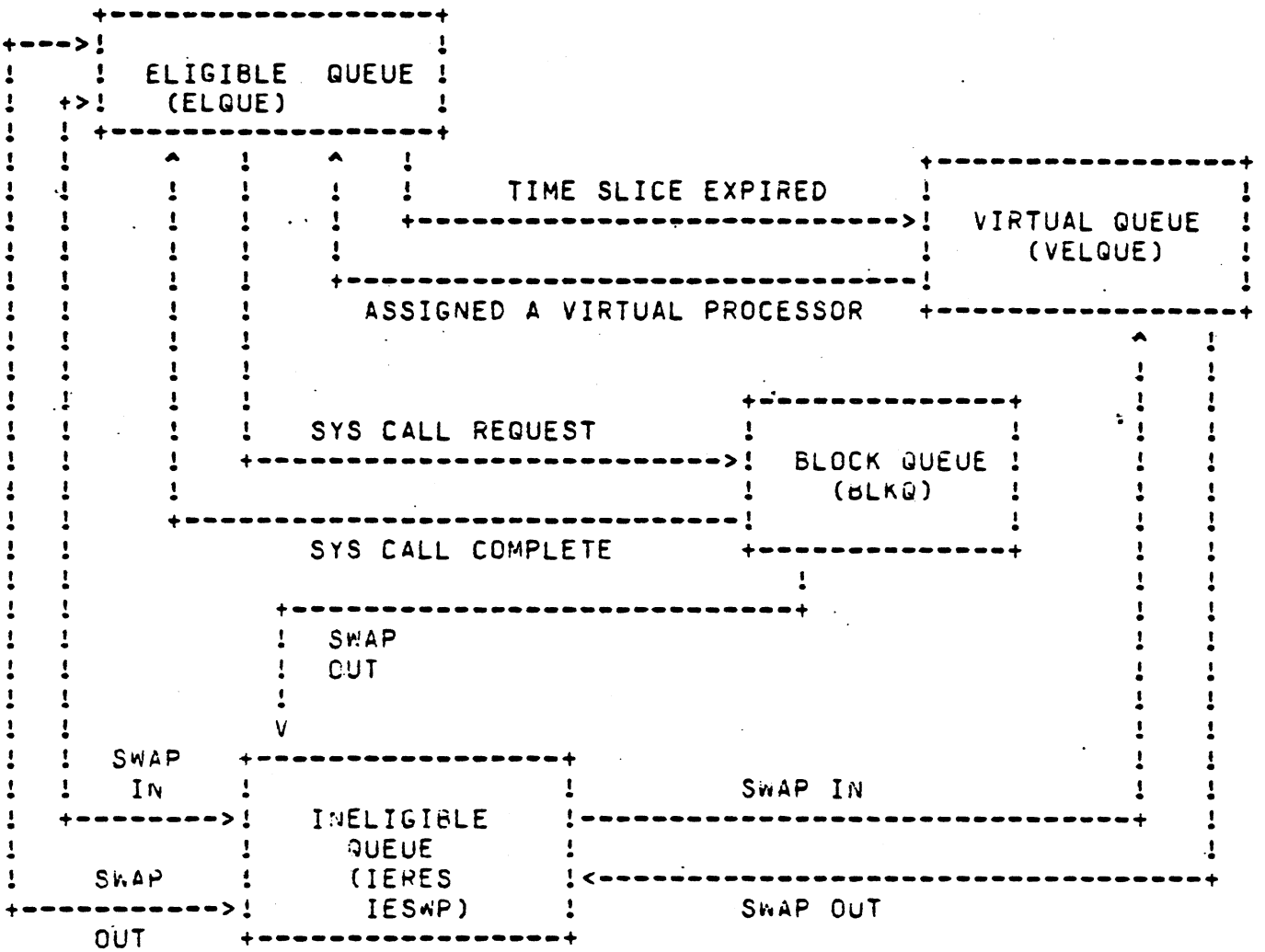
IESWP: Queue of ineligible swappable processes

IERES: Queue of ineligible preemptible processes

BLKQ: Queue of blocked processes

Core Manager queue: Queue of processes waiting for swap in or out

********** STATE-DIAGRAM FOR PTABLES **********



QUEUE VS. PROCESS TYPE RELATIONSHIPS

			RESIDENT	PREEMPT	SWAP
QUEUE	1.	ELQUE	X	X	X
	2.	VELQUE	X	X	X
	3.	IESWP	-	-	X
	4.	IERES	*	X	-
	5.	BLKQ	-	X	X

* RESIDENT PROCESSES ARE INELIGIBLE IN TWO CASES:
1. WHEN CREATED BEFORE MEMORY IS ALLOCATED.
2. WHEN A PREEMPTIBLE OR SWAPPABLE PROCESS CHANGES TO RESIDENT.

.2.3 CPU time contention

In general, AOS will allocate CPU time in the following overall priority structure.

Interrupt driven control functions that process events, these include:

Interrupts
Time Slice Completions (PIT interrupts)

System Control blocks, these include:

The Core Manager -- permanently the highest priority all other system daemons, and user system calls in a FIFO order.

Bound processes ordered by priority enqueue factor (PNQF)

RESIDENT & PREEMPTIBLE: PNQF = assigned priority
SWAPPABLE: PNQF = derived priority

RESIDENT and PREEMPTIBLE processes receive control for a fixed time slice of 2.048 seconds. When this slice expires, the process is re-linked to the end of its priority group. If there is only one process at this priority level, then the same process is run again for another 2.048 seconds. Every 32 milliseconds (a sub-slice), the process's tasks are rescheduled. This is done to allow a round robin scheduling of multiple tasks at the same priority level.

Swappable processes receive time slice based on past behavior and assigned priority.

T = time-slice = (32 millisec) * (2 ** S)

where: 1 <= S <= 6

Initial S is based on program size:

If the program size is less than 24 kw, S = 1;
If the program size is greater than or equal to 24 kw, S = 2.

The initial S will then be modified based upon the use of the allocated time slice by the process.

If the process blocks before the full slice expires, S will be given a new value based on the number of subslices used.

If process is still running when the time slice expires, S will be incremented by 1 the next time the process is scheduled. However, if the swappable process' priority is > 1, and the current S is 6, no change will be made to S. If the swappable process' priority is 1, S may reach an effective value of 7 (S will still = 6). In this case, a compute bound, or non-interactive, swappable process of priority 1 can attain a time slice of 4.096 seconds. It will, therefore, receive twice as much CPU time as an equally compute bound swappable process with a lower priority.

Actual time slicing is done on 32 ms. intervals. Hence, 2 ** S yields the number of sub-slices.

The scheduler maintains a count of the subslices for each process and the remainder of any incomplete sub-slice in the process's process table. This is done in case the process is pended due to:

1. Processing of an interrupt.
2. The scheduling of a system control block, or a higher priority process after an interrupt.

Control will return to the interrupted swappable process when swappable processes are again allowed CPU time.

Priority enque factor derivation for swappable processes

- The basic equation:

  PNQF = (slice-exponent) * 4 + 180 + PRI

Notes:

- PNQF = 100000 if the process is terminating.
- 180 is set to insure that all swappable processes have a lower priority than resident and preemptible processes.
- The priority is assigned by system manager or the user. AOS never changes the priority. (the values are 1, 2, or 3)
- The 'slice-exponent' is the S explained earlier.
- PNQF values are read as unsigned values. The lowest PNQF values have the highest priority for CPU time.
- The swappable process priority only serves as a fine control for scheduling swappable processes with equal time slice exponents. The priority forms the least significant portion of the PNQF word.

- If the PNQFs for two processes are equel, they are managed on round-robin basis to insure all processes can get CPU time.
- The PNQF is updated whenever a process blocks, or a time slice expires.
- If a process blocks before the end of its time slice the PNQF is immediately assigned the lowest allowable value (highest priority)

4.2.4 The AOS process scheduler

The following is a trace of the code path involved in scheduling AOS processes and control blocks. Remembering the layout of ELQUE (which along with VELQUE are the really important scheduling queues) all CBs (PNQF=0) precede all PTBLs. This will force the scheduler to first attempt to run any CB and then attempt to run each PTBL it encounters.

1. Set SYSIN = 1 (indicates that we are in AOS)
2. Enable interrupts
3. Set up the processor stack by loading the SP, FP, stack limit and overflow address.
4. Enter into the disk world and run any readied IOCBs.
5. Check to see if a subslice expired. If so, jump to TSPRC to process.
6. If the ASYNC DCU needs servicing (DCUWK<>-1) ready the PMGR and enable rescheduling of its tasks.
7. Scan ELQUE looking for a CB or PTBL ready to run. (After a powerfail all PTBLs are effectively unable to run). If we find one, run it by executing a jump @ through offset PPC of the CB or PTBL. PPC will contain either the location of TACT(the CB startup routine) or PCALL (the PTBL startup routine). These routines are described below.
7. No readied PTBLs or CBs have been found. Enter into the checksum loop. The checksum loop performs two checks on lower page zero constants.
   a. It reads each location twice. If the values are different, PANIC 11002. This can occur if the hardware is flaky or if the checksum loop was interrupted and page zero was modified during the interrupt processing.
   b. AOS performs a true checksum on the values. If the value of the checksum is wrong, then PANIC 11001.

8. If during the checksum loop, some process had been readied, the interrupt world would have reentered the scheduler at SMON1, or (3) above. Therefore, at this point, there is still no process or CB ready to run.

9. Multiplex: These routines are designed to make sure that processes on VELQUE will work their way onto ELQUE. This is done in the following manner.

   a. MPLEX first calls MKEL to see if anyone on VELQUE can be moved onto ELQUE. If not, return to the scheduler at SMON1 (3) above.

   b. If someone is ready to join ELQUE, MPLEX tries to find a free virtual processor (VP). The easiest way to get a VP is to find one on the free VP chain. If there is one, MPLEX takes the entry on VELQUE, binds it to the virtual processor, and puts it on ELQUE (in the proper PNQF place).

   c. If no VPs are on the free chain, MPLEX will attempt to find one that can be unbound from ELQUE. It does this by calling MKINEL. If all processes on ELQUE have their do not unbind bit set, take the error return (which will take us back to SMON1 i.e. try again later). Next we try and find the lowest priority process that can be unbound, and for some reason cannot run. If any exist, this is our candidate. If no process meets the above criteria, then MKINEL will return the lowest priority PTBL on the queue as the best candidate.

   d. MPLEX will take the process off VELQUE and enqueue it to ELQUE in PNQF order. It will take the other (possibly reluctant) process that was on ELQUE off, and put it on VELQUE (also in PNQF order).

   e. Finally, MPLEX returns to the scheduler which jumps immediately to SMON1 (point 3 above). TAH DAH !!

At this point, it is assumed that the scheduler has found either a process table or control block ready to run. The scheduler has done a JMP @PPC through offset PPC of the current PTBL or CB, which contained one of the two values:

   a. TACT - Control block startup. The readied entity on ELQUE was a control block.

      1. Decrement the current event counter (UPQUE) (used by other routines to determine at what point to reenter the scheduler)

      2. Set up the control block's stack

3. Restore map slots 60, 62, and 74

4. RTN. Since we switched stacks, this instruction will return us to the pend location for this CB.

b. PCALL - Process table startup. The readied entity on ELQUE was a process table.

1. Set up the per process stack.

2. Perform the following operations in this order:

a. SCHED ACTION --	Don't touch this process. (JMP RUNEX)
b. OP INTERRUPT --	Startup term daemon.
c. UNPEND PROCESS --	reserved for future use.
d. SWAP OUT PROCESS --	Calls PRBAG in COREM to enqueue process to core manager for swap out.
e. BLOCK PROCESS --	Calls CTBLK in COREM. Then schedule next PTBL on ELQUE (SMON2)
f. START UP DAEMON --	Attempt to allocate stack and CB. If these fail ignore process (try again later). If good, set DON'T MAKE VIRTUAL and SCHED ACTION bits.
g. WAIT MKEY CHANGE --	If MKEY changed (AOS available memory has grown), the process can be started again, else ignore this process.
h. TIME SLICE END --	Go to TSUP in COREM. Shuffle processes on queues (if possible) Set AT LEAST 1 TIME SLICE END bit in PTBL.

3. If none of the above operations are required, then process system calls outstanding for this process. This is done by jumping to PCBST which is a bridge between the AOS scheduler and the user scheduler.

PCBST: This routine will process any TCBs with outstanding system calls. These TCBs will be enqueued off the PEXTN at offset PSWD The following logic is performed in PCBST.

a. Check to see if the scan of backed TCBs is inhibited. If so JMP NOREQ (no request).

b. If no system calls are enqueued go to NOREQ.

c. Check system call validity. If out of range -- error Illegal command (1), make the next TCb first on list, and jump back to PCALL.

d. If system call requires a CCB (channel call) and the CCBs must be faulted in, start the appropriate daemon.

e. If call is parallel, and a call is in progress go to NOREQ.

f. If a direct call, go process (JMP DIRRS in SCPRC).

g. Attempt to allocate stack and CB. If none available go to NOREQ. (Note that this implies that if any TCB behind the first can run, it will not because the first cannot.)

h. Unlink this TCB from the chain and if necessary map the CCB.

i. Set up the control block stack and enqueue the CB to ELQUE.

j. Set the DON'T MAKE VIRTUAL bit in the PTBL, increment the active call count (offset PSWCT of PEXTN).

k. Jump into the appropriate system code path.

NOREQ:

Check to see if the DON'T ENTER bit in the process table is set. If it is, then that means that the last time we scanned TCBs there were none ready to run, and that nothing has happened in the mean time to ready any. So just jump back to RUNEX (try again later...). If not, then we are ready to enter the AOS task scheduler.

4.2.5 The AOS task scheduler

The AOS task scheduler schedules user tasks for a specific process table. The following is a description of the code in PSCHED.

1. (PSCHED) If schedule inhibit is set in UST then jump to PSCDR.
2. Find a ready TCB. If there are none then go to PSCDR.
3. Round robin the TCBs in the found TCB's priority.
4. Set up the TCB's floating point unit, extended variables and stack.
5. If RESCH or UPQUE is non-zero, an event has occured that has made a control block ready to run. Go back to the scheduler which will find the readied CB. (So close and yet ...)
6. Set up the map for the task.
7. Start the PIT.
8. And now we let the user run.

PSCDR:

At this point, AOS has found no active TCBs ready to run. If the TASK RESCHEDULE bit has been set, then the interrupt world has completed some action that might have readied a task so we jump back to PSCHED.

If the process is totally idle (no outstanding TCB requests, and the process has not issued a serial call) then block the process.

Schedule the next process table. (SMON3 or SMON4 in SCHED).

4.2.6 PEND / UNPEND

There are two additional routines found in SCHED that are relevant and independent.

PEND:

This routine is used to pend a CB's code path. It is called with a JSR @.PEND.

1. SAVE 0.
2. Store the unpend key in offset CKEY in the CB.
3. Set the PEND bit
4. Jump to RUNEX (which will reset the RUN bit, and jump into the scheduler which will run the next entry on ELQUE).

NOTE: we initially do a SAVE, but execute no return. This will leave the return address (unpend address) on the stack. To return to the place were we pended, simply restore the CB stack and RTN. (this is done by TACT)

UNPEND:

This routine will unpend any control block on ELQUE that is waiting for a specific key (passed to the routine). It does this by resetting the NOT READY TO RUN bit. UNPEND will also increment UPQUE for each control block unpended.

4.2.7 Event Synchronization

It is unusual for AOS to process for an extended period of time. Usually, many paths require short periods of CPU time. When a path gives up control, it is generally because it's waiting for an event.

Typical events are:

1. Waiting for a disk block to be read into AOS buffer (e.g. opening a file requires the reading of a directory entry.
2. Waiting to access a database being used by another active path.
3. Waiting for a task's system call to complete
4. Waiting for an AOS global resource.

The four events mentioned, can all be experienced in a single system call. Each Control Block and Process Table have an event key called 'CKEY'. The general procedure followed is:

1. The path calls the subroutine 'PEND', passing to 'PEND' the key it wishes to wait on.
2. The path becomes quiescent (i.e. it is not given any CPU time).
3. Another path calls the subroutine UNPEND, passing to 'UNPEND' the key that the first path is waiting on.
4. The waiting path now becomes ready and resumes execution.

All paths waiting on given key are readied when 'UNPEND' is called.

Exception to key method: If key = -1 unpending occurs by direct manipulation of the penced caller's process table's, or system control block's, status bits. in this case:

1. Unpender knows what database to unlock, and

2. Avoios the overhead of a linked list search.

Types of PEND keys

1. Data base address (KEY > 400)

2. Special keys

SKTRM = 1	Wait for son proc term
SKOOL = 2	Overlay is loading
SKOOV = 3	Overlay pool needs to grow
SKOOM = 4	AOS needs memory
SKSwP = 5	Wait for swap of proc to complete
SKSIO = 6	Shared read wait
SKBUF = 7	Base level AOS needs buffer
SKNGP = 8	Wait for I/O NQ block

AOS uses hierarchical event locking. A majority of the paths only lock a single database. If two databases must be locked, then a path requiring the locking of database 'A' and then database 'B' will require that any path that uses these two databases must lock them in the same sequence.

4.3 COREM and its associated routines

The AOS routine COREM is responsible for managing overall memory requirements; pending users waiting for memory and growing overlay and buffer pools. In addition COREM handles system unit errors. Not all of the code in the COREM modules is directly related to the COREM code path. A few other routines will be documented below.

4.3.1 COREM introduction

The core manager runs as the highest priority control block on ELQUE. It remains dormant until a code path calls the routine CWAKE which sets the ready to run bit in the status word for the CB in accition to the words or bits needed to indicated which action COREM should process when it gets control of the CPU.

4.3.2 COREM code path

1. Decrement UPQUE (the number of CBs waiting for processing).

2. Set up PPC of the core manager's CB to point to TACT in case we pend. When the core manager is about to go to sleep, PPC will be set to CMINT, COREM's initialization code.

3. Assign the root process table as a dummy process table for routines that require one.

4. First process swaps. Look at CMQWD (offset to chain of PTBLs needing swap in/out). If the chain is empty go on, else process each swap in or swap out. (call resident overlays SwAPI or SWAPO).

5. No more swaps, see if a system overlay block can be deallocated. If it can, do so (FREOV in CMOV1). Overlay blocks can be released if:

   a. The minimum number of system overlay blocks is satisfied.
   b. The overlay block is free.
   c. The length of time since the overlay was lasted referenced exceeds the minimum time that an overlay must be kept in core.

6. See if a system buffer can be released, and if so, do it (FREBF in CMOV1). This is possible if:

   a. The minimum number of buffers have been met.
   b. The number of buffers on the free LRU chain exceeds the number of buffers in use.

7. Process any special requests to COREM (flag word is SMFLG)

   a. 1b0 -- Unpend resident processes waiting for memory release. This is done by loading the appropriate key and calling UNPEND.

   b. 1b1 -- Grow system buffer pool. (calls BFGRW in CMOV1) If we are in system initialization inhibit the grow if the max has already been allocated.

   c. 1b2 -- Grow system overlay pool. Allocate header. Allocate memory page. (If no memory available, release header and try next request). Set up the header and unpend those waiting for overlay load.

   d. 1b3 -- Scan the block queue for anyone waiting to unblock. (call BSCAN in CMOV1) A process will not be unblocked if it is waiting for son's termination or it has the master block bit (?BLKPR) bit set.

   e. 1b4 -- No longer used.

   f. 1b5 -- Report unit error. (Call overlay UNERR)

   g. 1b6 -- Shrink size of shared page candidate chain.

8. See if any processes on the ineligible queues can be swapped in.

9. Check to see if any core manager request have come in while we were running. If so start the loop again. (this is done with interrupts disabled). See 4 above.

10. Put the core manager to sleep (put CMINT into PPC, clear CRSEG (366) and go process next CB (or PTBL) on ELQUE.

4.3.3 SWAPI (Swap in logic)

The SWAPI overlay handles the shared portion of a swapped out process. It is called from COREM with AC2= PTBL of the process

SWAP IN (SWIN)

This primitive begins the swap in of a swapped out core image; the extender and shared area will be swapped in here and a chain to the common swap in/out primitive SWAPM (entry CCBIO) will finish. The core manager determines that an enqueued process needs to be swapped in and calls this primitive. The swapped out image contains the following information :

- The Process Table Extender
- The Shared Directory for the Primary and Ghost
- The CCB'S (0 to 4 256. word pages)
- The Primary and Ghost unshared areas

The process extender is read in and moved into a quarter page associated with the process. If a shared area exists, the shared directories are read in and each shared slot is processed. The shared slots have the following potential usage:

- Idle
- Remapped to the primary (if a GHOST slot)
- An active shared page

Shared pages are looked for on the FCB chain the slot is associated with. If they are found, the page is put in the process context & the use count increment. The pages not found are read in & placed on the FCB chain in the manner that an ?SPAGE would. An attempt is made to read a contiguous group in order to minimize disk latency.

If the process being swapped in is in a first load mode, then only the extender is read in. The rest of the image will be read in by a daemon.

Since this primitive will grow a processes memory holding by doing shared reads & picking up quarter pages for CCBs, memory preempt is possible. If this process can't preempt, the swap in is aborted & the process goes back to the ineligible queue to try again later.

Notes: This primitive is only called by the core manager and assumes that the unshared area has been picked up and that there is enough memory for any shared pages that need to be read in. This determination was made before the process was enqueued to the core manager.

The FCB associated with a shared page is locked to prevent a race with an SPAGE for the same page.

4.3.4 SWAPO (Swap out logic)

The SWAPO overlay swaps out the shared portion of a process. It is called from COREM with AC2= PTBL of the process

SWAP OUT (SWOUT)

This primitive is called by the core manager to swap out a process the extender and shared area will be swapped in here and a chain to the common swap in/out primitive SWAPM (entry CCBIO) will finish. The swapped out image contains the following information :

- The Process Table Extender
- The Shared Directory for the Primary and Ghost
- The CCB'S (0 to 4 256. word pages)
- The Primary and Ghost unshared areas

The shared directory will be created and written to disk if the shared area changed flag is set & a shared context exists for either the primary or ghost. The directory shows shared slots in the following states :

- Idle
- Remapped to the primary (if a Ghost slot)
- An active shared page

The directory contains information for swap in which tells it how to restore each shared slot to its current usage. If the shared area changed bit is reset, than the last directory written out is still valid for both contexts.

The shared use count used by the system to estimate the sharing incidence will be derived. When each shared page is processed the use count will be looked at. If the count is = 1 (we are the only user), then we will assume that the page will be reloaded at swap in time. The total count is kept in the process table offset PSHFC.

Notes: This primitive is only called by the core manager. The process that is being swapped out is on the eligible queue with the wait bit (PSEWB) Set (this prevents any other action from occurring).

In a modbits system, any shared page that has it modified bit set in the process table must also set the bit in the CME for the hardware page modified flag.

4.3.5 SWAPM (Swap in/out completion)

The SWAPM overlay swaps in/out the CCB'S and the unshared portion of a user. The stack has been set up by SWAPI or SWAPO, from which this overlay is chained to.

This primitive completes the swap in or out of a process. It is called by SWIN or SWOUT. The description will be broken out for swap in and out, code paths are the same with minimal branching.

Swap In:

The CCB'S are read in if they are defined and if the flag for use during the last eligible state is set. If the user needs them and they are not read in here, they will be loaded by a daemon when needed. A quarter page is picked up for each CCB group defined. If a quarter page can't be allocated, the swap in is aborted and retried later. The unshared areas for the primary and ghost are read in in one I/O request. This is done to minimize disk latency. The process map is "shuffled" to make both unshared areas contiguous and then restored. When the swap in is completed, the process is readied. Anyone waiting on the swap in to complete will be unpended.

Swap Out:

The CCB'S are written to the swap file and the quarter pages containing them are released. The CCB slots are marked as faulted out. The unshared areas for the primary & ghost are written out in one I/O request. This is done to minimize disk latency. The process map is "shuffled" to make both unshared areas contiguous and then restored. When the swap out is done, the bits set to inhibit the process from running during the swap out are reset. The process is enqueued to the appropriate ineligible queue (swappable or preemptible) unless the process is blocked (it already is on the blocked queue). Anyone waiting for the swap out to complete is unpended.

Notes: There is a special entry into the swap out code (SWIAB) for swap in attempts that abort. This entry effects a swap out and releases any resources held by the swap in such as memory

This routine assumes that it is only chained to by swap in or out and that all temporaries set up by the caller.

4.3.6 TSPRC / TSUP

These routines process time slice end. The processes exponent will be incremented (if possible), a new PNGF computed, MPLEX will be called to see if can do anything (see scheduler documentation), and the core manager will be awaken (CWAKE) to see if anyone can move off the ineligible queues. See section 4.5.1 for a complete discussion of full and sub slice processing.

4.3.7 PRBAG

This routine marks a process for swap out. If the process does not have the "do not make virtual bit" set, then set the swap direction indicator to 'out', unbind the process from ELQUE and enque it to both VELQUE and the core managers swap queue.

4.3.8 MSOLV

Process a request for more memory by attempting to preempt one page of memory. If the preemption is possible, pend until complete and take good return. If not possible and non resident, set up the process to swap out. If resident wake up the core manager and hope something happens.

4.3.9 Memory Pre-emption

Memory pre-emption as a function of process type/runtime characteristics.

Key: R -- Resident P -- Pre-emptible
I -- Interactive N -- Non-interactive

	R	P	I	N
R	NO	YES	YES	YES
P	NO	YES (1,2)	YES	YES
I	NO	YES (2)	YES (2)	YES (4)
N	NO	YES (2)	YES (2,6)	YES (3)

Notes: Read the above chart as "LEFT" can pre-empt "TOP"

1. A pre-emptible process will pre-empt a pre-emptible process only if its ?PNGF is lower.
2. This pre-emption can happen only if the pre-empted process is blocked.
3. Only if the time slice is up, and the ?PNGF is lower.
4. Only if the total eligible non-interactive swappable processes is > than the low value bias factor.
5. A comma in the above notation (1,2) means 1 "or" 2.
6. Only if the total eligible non-interactive swappable processes is < than the high value bias factor.

4.4 The interrupt world

When an I/O device completes its operation and is ready to receive/send more data, it requests an interrupt. As soon as the CPU is at an interruptable point in its processing, and has finished servicing data channel requests, it takes care of the interrupt.

Upon servicing an interrupt, the Eclipse CPU does four things :

- disable interrupts,
- disable the map, which means addresses are simply translated logical to physical,
- save the updated current program counter in location zero,
- jump indirect to location one, where it expects to find the address of the interrupt service routine.

After servicing the device, the interrupt service routine should re-enable interrupts and immediately jump indirect to location zero. The hardware will allow this next instruction to be executed in any case, even if another interrupt has come in. This ensures proper return of control with nested interrupts.

If interrupts are disabled throughout the interrupt service routine, the CPU can no longer be interrupted until this device servicing is finished, and all other devices requesting interrupts must wait. This might lead to losing data on fast unbuffered devices. Therefore more sophisticated hardware instructions are available to implement a system of interrupt priorities which will permit some devices to interrupt others.

Every I/O device is assigned an interrupt mask bit by the hardware, and the interrupt service routines can control interrupt priorities by setting interrupt masks : any device which should not interrupt the device being serviced is masked out (prevented from requesting an interrupt) if its mask bit is set. The mask bits corresponding to devices which can interrupt are zeroed. By changing the priority mask, an interrupt service routine can mask out those devices whose interrupts are undesirable, without disabling interrupts for the duration of the service. The hardware reserves location 5 for storing the current interrupt mask.

For more details on the above, the devices mask bit assignements and the interrupt related instructions, see an Eclipse "Principles of Operation" manual.

4.4.1 AOS handling of interrupts

Location 1 of a machine running AOS will always point to INTS, the interrupt dispatching routine. Control is transferred by the hardware to this module every time we get an interrupt. The interrupts will be dispatched using the 'VCT' instruction, according to the interrupt vector table BTBL in the system upper page zero (STABLE). BTBL is 64. words long and each entry is first assembled as pointing to the undefined interrupt service routine IUD. SINIT fills in BTBL with the actual DCT addresses using the table of devices built by AOSGEN; it leaves the entry as is, i.e. pointing to IUD, for all device codes not in use.

When a user IDEFs a device, the IDEF routine will put the location of the user DCT in BTBL.

All defined interrupts will vector to the routine MAPST, the address of which is stored in bits 1-15 of the first word of the DCT; this is to save the current state of the map before jumping to the real interrupt service routine for the device, pointed to by the fourth word of the DCT.

All entries in the interrupt vector table except the ones pointing to IUD have the "push bit" (bit 0) set in the DCT (modes C and E of the VCT instruction). All undefined interrupts will vector to IUD directly (mode A of the VCT instruction).

Before branching to MAPST, the VCT instruction (modes C and E) pushes the current interrupt mask at location 5 onto the stack, does a logical 'OR' of this mask with the second word of the DCT, places the result in location 5, issues a mask out with this value and turns interrupts back on.

AOS interrupt handler requires use of a stack. The 'SS' stack is used whenever AOS is not at base level. This means that a VCT with stack change bit set (mode E) is issued when an interrupt comes in at base level, and a VCT without stack change bit set (mode C) issued when we are at interrupt level. The interrupt flag INTLV is initialized as minus one at base level and incremented for each nested interrupt.

AOS dismisses a defined interrupt by executing an SVC instruction which will jump @2. Previous to servicing the interrupt, MAPST had placed the location of 'DISMIS' in location 2. DISMIS is the common interrupt dismissal routine. In the case of an undefined interrupt, IUD exits the interrupt world by re-enabling interrupts and issuing a JMP @0.

4.4.2 The INTS module

The INTS module has the following entry points :

INTS	interrupt service
MAPST	map save routine
IUD	undefined interrupt handler
OVFLO	interrupt stack overflow handler
SOVF	system stack overflow handler
PFL	power fail (lack of) service
IERCC	ERCC error handler
IRTC	real time clock service routine
IPIT	programmable interval timer service routine
UDEX	enter user defined device
UINTR	return from user device driver
IWKUP	awaken a process
DISMIS	(not an entry point) interrupt dismissing routine

4.4.2.1 INTS logic

INTS is the interrupt dispatch code. It reads out the current map, saves it for MAPST, enters the 8 map so that DCT's can be accessed, increments the interrupt level flag INTLV and issues a VCT instruction with stack switch if INTLV is now zero, without stack switch if INTLV is positive. The VCT dispatches on the interrupt table BIBL in page zero.

4.4.2.2 MAPST logic

All defined interrupts vector to MAPST which saves map constants before going to the real interrupt service routine for the device. The contents of the following locations are pushed on the (interrupt) stack, in this order :-

- location 2
- MAPSV (map state saved by INTS before the VCT)
- CUR60, CUR62, CUR64, CUR66, CUR74
- CRSEG (current overlay running)

If we are at the first interrupt level the PIT is read, turned off and the value saved in page zero. Location 2 is set up to point to the DISMIS routine, and control is transferred to the device interrupt service routine.

4.4.2.3 IUD logic

IUD processes all undefined interrupts, i.e. interrupts coming from devices not sysgen'ed. It issues an INTA to obtain the device code of the interrupt. If the DCT for this device code is undefined IUD will try 2000. times to clear the interrupt, after which it will panic. If the interrupt clears the map is restored, interrupts re-enabled and a JMP @0 executed.

If the DCT for the device code is defined, then IUD tries re-processing the interrupt by jumping to INTS.

4.4.2.4 OVFLO and SOVF logic

These two routines handle stack overflow conditions by setting up values for the panic message and jumping to the panic routine. If it is a system stack overflow the current frame pointer, stack pointer and stack limit are recorded. For an interrupt stack overflow the same plus the last interrupting device's code and the value of the interrupt level flag are saved.

4.4.2.5 PFL logic

This routine is jumped to by MAPST, with interrupts off, on a powerfail interrupt when no power fail recovery has been sysgen'ed. Upon detecting a loss of power, the power fail facility on the Eclipse sets the power fail flag to one and requests an interrupt. The power fail facility has no mask bit in the priority mask, and responds to an INTA or VCT instruction with device code zero.

PFL checks the power fail flag. If zero the power fail is not real and we just increment the counter of interrupts from device zero, PCNT in STABLE, (or panic if PCNT overflows) then dismiss the interrupt via an SVC.

If the power fail flag is set we did loose power. As the power fail restarts by executing the contents of location zero, we store there a JSR @.PNIC followed by the appropriate panic code, and HALT the CPU.

4.4.2.6 IERCC logic

Upon detecting a memory error, the ERCC facility generates an error code and requests an interrupt. The ERCC facility has no mask bit in the priority mask.

IERCC is entered by MAPST with interrupts off. It saves the memory fault address and fault code in the DCT, checks if the error is correctable, panic'ing if it is not.

If the error is correctable, the ERCC facility is re-enabled, the core manager is flaged to call SYSERR and log the error, and the interrupt is dismissed.

4.4.2.7 IRTC logic

This is the real time clock interrupt handler. Every clock tick it performs the following :

- if there are synchronous lines, the synch line timer is decremented; if it goes to zero IRTC branches to the synch line time out routine (see chapter 12).
- if there are any processes with histograms, the histograms are updated.
- the time remaining for any process on the delay chain is decremented; if it goes to zero and the process is eligible, the task making the delay call is unpended.

These last two functions are performed in a resident overlay (RTCIS in DVOV6) jumped to by IRTC.

Every second IRTC performs two other things :

- update the time of day TODL/TODH in SZERO.
- scan the device time out queue TOQUE in STABLE, decrementing the time remaining for the device to be on the queue; if the time remaining goes to zero, then the time out routine for that device is entered. TOQUE is a queue of time out blocks built at system initialization time, one block per magtape or disk DCT.

4.4.2.8 IPIT logic

The contents of the PIT has already been saved by MAPST, hence IPIT just dismisses the interrupt after forcing a reschedule.

4.4.2.9 UDEX and UINTR logic

UDEX is entered by MAPST for all user defined devices interrupts. The user map is loaded, the needed context mapped, location 2 set up with the address of UINTR and the user entered via a JMP @0 .

The user returns with an SVC, which will transfer control to UINTR. If no reschedule is requested the old process' context is restored and the DISMIS code entered. If user asked for a reschedule the routine UIWKUP is executed before dismissing the interrupt.

4.4.2.10 IWKUP and UIWKUP logic

IWKUP readies a specified process by setting to zero the "Not ready to run" and the "No task ready" bits in the process table status word. If called from within the interrupt world, it also sets to one the "Reschedule" bit, and if the process is resident but not the PMGR it increments the reschedule flag 'RESCH' so that the dismiss code will reschedule. IWKUP re-enables interrupts.

UIWKUP is a special version of IWKUP called from UINTR. It readies a specified process to run by resetting the same bits as IWKUP, but does not enable interrupts, as they must stay off while processing dismissal of a user interrupt. UIWKUP also takes advantage of the fact it is called from the user interrupt service code and thus knows the process to be readied is resident and not the PMGR, to skip these checks.

4.4.2.11 DISMISS logic

This is the common routine to dismiss all defined interrupts. It is jumped to through location 2 via the SVC instruction, which means the map has been disabled. The current stack is the interrupt stack and the following information was pushed on it :

CRSEG
CUR74
CUR66

CUR64
CUR62
CUR60
pre-interrupt map status
pre-interrupt contents of location 2
return block pushed by VCT instruction

The first thing DISMISS does is to disable interrupts, so as not to be disturbed at a delicate moment. Next the top eight items describing the pre-interrupt state are popped. The B map is reloaded and the window slots which the interrupt world might have used (the CURxx above) are restored. The interrupt level indicator INTLV is decremented, the mask pushed by the VCT instruction is popped, stored in location 5 and a MSKO issued with it. If dismissing the present interrupt does not bring us to the base level, we re-enable interrupts and pop a return block, thereby falling through to the previous level.

If we are dismissing a base level interrupt we will exit the interrupt world and have a lot more things to do. First, location zero is set up for catching the "JMP 0"'s, by storing an SCL into it (in case of a JMP 0 this will push a return block on the stack with PC=1, which SYST will then detect). By looking at the map status at the time the interrupt was taken, we see if a user or AOS was interrupted.

If AOS was interrupted in its idle loop, we check to see if the interrupt has set the reschedule flag or brought a control block request. If so we exit the idle loop and enter the scheduler for a scan of ELQUE. In this case the return block pushed by the VCT instruction is not popped as we are now at base level and the interrupt stack will be reinitialized from locations 4,6 and 7 - which never change - when the next interrupt comes in. If AOS still has nothing to run we execute a RSTR which brings us back to the idle loop.

If AOS was interrupted outside of the idle loop, we just return to whatever we were doing by issuing a RSTR. Note a RSTR switches us back to the stack in use at the time the interrupt came in.

If a user was interrupted, the reschedule flag and control block request queue are checked. In the case the interrupt did not bring anything to reschedule, we go see if the user's time slice has expired. If not, the PIT is restarted, the map state restored and an RSTR issued, which transfers control back to the user.

If a reschedule is required or the user time slice has expired, the A map is pulsed twice to get the user's USP and scheduler mode, the B map is entered and the user's page zero mapped to slot 76000. If the time slice has not expired, the running bit in the process table status word is reset and the PIT residue, task, user stack and FPU states are saved in the extension before entering the scheduler at SMON.

If the user has exhausted his time slice (actually sub-slice), DISMIS checks to see if his CPU time allowance has been exceeded. If so the termination bits are set, if not the count of sub-slices is updated. In any case the scheduler is entered at SMON after saving the task state as above.

4.4.3 Other interrupt world modules

Most device interrupt service routines are defined out of the module INTS. Some are part of the AOS kernel, some are in resident overlays with just a wart in the kernel to perform the overlay call and branch to DISMIS - via an SVC - on return from the overlay. Besides, the interrupt world calls miscellaneous routines which are in resident overlays, like PBITS in RROV2 to set the termination bits of a process having exceeded his CPU time allowance.

Below are listed the interrupt service routine entries for the devices that AOS supports, except the disk interrupt service routines which are mentioned in chapter 5. For further details on a specific device interrupt service, find a copy of the "Peripherals" manual and refer to the code itself.

In PERDR :

TISER teletype input interrupt service

CISER other non-multiplexed character devices input interrupt service

COSER all non-multiplexed character devices output interrupt service

In CDRDR :

CDRIS card reader interrupt service

In MUXDR :

MUXIS asynchronous line multiplexor interrupt service

In SLMIS :

SLMIS interrupt service for the synchronous line multiplexor, handles transmitter, receiver and modem interrupts.

In DCUIS :

SDCIS synchronous DCU interrupt service routine, re-entrant so as to handle up to four DCU's. SDCIS is entered when a DCU resident character routine enqueues a command onto the slave-to-host queue and sets the DCU flag, thus causing a host interrupt (see chapter 12)

In DCUDR :

DCUIS asynchronous DCU interrupt service

In IOPDR :

IOPDR IOP interrupt handler. If DONE(IOP)=1 then it is an IOP programmed interrupt. If DONE(IOP)=0 then it is an IOP memory parity error. In either case the IOP will have halted.

In DVOV1 :

RTAIS magtape interrupt service

In DVOV5 :

RCTIS MCA transmit interrupt service

RCRIS MCA receive interrupt service

In DVOV6 :

RTCIS real time clock interrupt service finish up code, entered from IRTC in INTS.

In LP6DR :

RPRIS interrupt service for a data channel line printer, entered for every controller status change and whenever an I/O request has finished.

RPDIS same as above but for a data channel LP2 .

4.5 Miscellaneous

This section will discuss some of the AOS routines and concepts that can not be classified as exclusively part of one module.

4.5.1 AOS Timeslices

There are two types of timeslices under AOS. The first, the subslice (Ss), is always 32 ms long and is defined by the interrupting of the PIT. The second, the user's timeslice (Ts), is defined as:

$$Ts = Ss * (2 \hat{} S)$$

where S (varying between 1 and 6) is determined by the interactiveness of the user.

Subslice (Ss) end (PIT interrupt)

At the end of a subslice (PIT interrupt), the following occurs:

1. Bump the number of subslices run since last bound to a virtual processor (PSSEL in PTBL)
2. Increment the user's CPU usage by 32 ms.
3. If the user has resource limiting (max CPU time) check to see if it has been exceeded. If so set the appropriate bits to force termination (TIME LIMIT EXCEEDED).
4. Reset the subslice (-320 -> PSL)
5. Decrement the subslice count and put the PTBL in TSPTB. If the count is not 0, then we are not at a full timeslice end, so set the high order bit of TSPTB (time slice end flag) to 1 to indicate PTBL shuffle only.
6. Save the user's task state.
7. Rescan the eligible queue (jump to the scheduler).
8. The scheduler will see that the flag (TSPTB) is set and will call TSPRC below.

Time- or Sub- slice end (as handled by the scheduler)

1. The scheduler checks TSPRC. It will contain either 0 (no slice end, a PTBL (timeslice end), or 100 + the PTBL (subslice end). We are concerned with the latter two possibilities: the scheduler will jump to TSPRC.

2. If this is a timeslice end (OBO), and the process is swappable then:

    a. increment the exponent
    b. calculate the new PNQF for the process.

3. If this is a timeslice end, scan the ineligible queues for processes to be moved to VELQUE.

4. Move this PTBL to the end of its priority group on ELQUE.

5. Call MPLEX.

6. Jump back to the scheduler

4.5.2 The BIAS factors

Two locations in page zero, BIAS and HBIAS in STABLE, define the AOS low and high bias values respectively. The low bias is the minimum number of non-interactive swappable processes which AOS tries to keep in memory at all times. The high bias is the maximum number of such processes which will normally be kept in, although if more non-interactive candidates would fit in without pre-empting other processes, they will be allowed as well. By convention, a high bias value of zero means no maximum is enforced.

A non-interactive process in the AOS sense is a swappable process having a time slice exponent of 6. Resident and preemptible processes are not considered for bias purposes.

The following system modules reference the bias values :

SSOV5 - Implements the ?GBIAS / ?SBIAS calls.

SWAPO - When a process marked for preemption is non-interactive, the low bias is examined, and if ELNON < = BIAS only another non-interactive process can preempt this one. Also, if a non-interactive process wants to preempt and the high bias is already satisfied, it cannot preempt any process on the blocked queue.

CMOV1 - If there are no resident or preemptible process waiting to be run, we will try and load a swappable process. In the case where ELNON < BIAS - low bias not satisfied - we will try to find a non-interactive process. If there is none we take any swappable process. If there is one but it does not fit, we will not run an interactive process.

4.5.3 Daemons

A daemon can be considered an AOS initiated system call (as opposed to the user oriented or standard call). When AOS needs something done, and the code path required might pend, AOS will use a daemon for the processing.

Daemons are currently used for the following:

1. Loading user CCBs from the disk. This path can obviously pend waiting for the disk
2. Process initial load. Again the path can pend waiting for the disk
3. Process termination.
4. Process keyboard interrupts
5. Sync line handling

Daemons are started by setting the request daemon bit in PSTAT. They run off control blocks (like system calls). They can be identified by examining offset 0 of the CB. It will contain a 0.

The code for processing daemons is located in the module SCHED.

4.5.4 Process creation

Process creation is one of those AOS functions that is not the responsibility of anyone module or routine. It involves system call processing, COREM, daemons and other such things. The following is an attempt to follow a process creation from the time at which a process performs a ?PROC until the new process is healthy and strong, ready to assume its place in the AOS world.

?PROC Create a New Process

The ?PROC is a combination ghost/kernel AOS system call. The users ?PROC is first interpreted by the ghost which then builds an initial IPC message from the user packet that contains the names of the generic files @LIST, @INPUT, @OUTPUT, @DATA. This IPC message is sent to the new process to be picked up if and when the new process starts up a ghost. The ghost then issues a ?GPROC that actually gets AOS involved. AOS does the actual processing and then control is returned to the ghost for some post processing.

NOTE: From here on, ?PROC and ?GPROC are used interchangeably to mean the system portion of the ?PROC call.

SYSTEM PROCESSING OF THE ?GPROC CALL

Input to GPROC system call:

AC0	Length of initial ghost IPC message
AC1	Address of initial ghost IPC message
AC2	Address of ?PROC parameter packet

Parameter Packet entries:

0	?PRFLG	Flags
1	?PRSNM	Byte pointer to program file
2	?PRIPC	Address of IPC header
3	?PRNM	Byte pointer to process name
4	?PRMEM	Maximum memory size in blocks
5	?PRPRI	Priority
6	?PRDIR	Byte pointer to working directory
7	?PRCON	Byte pointer to console name
10	?PRCAL	Maximum number of active system paths
11	?PRUNM	Byte pointer to user name
12	?PRPRV	Process privileges
13	?PRPCR	Maximum number of sons
14	?PRIFP	Byte pointer to INPUT file
15	?PROFP	Byte pointer to OUTPUT file
16	?PRLFP	Byte pointer to LIST file
17	?PRDFP	Byte pointer to DATA file

Note: Offsets 14-17 are interpreted by the GHOST only

Process Creation Flags:

1B0	?PFPP	Proc a parallel process
1B1	?PFDB	Enter at DEBUGGER
1B2	?PFEX	Caller is blocked until son terminates
1B3	?PFPM	Mask off son's privilege
1B4	?PFPX	Indicates packet extension exists
1B5	?PFDA	Do not pass DACL
1B14	?PFRS	Resident process
1B15	?PFRP	Pre-emptible process

?PROC Parameter Packet Extension

36	?SMCH	Maximum CPU time (high order bits)
37	?SMCL	Maximum CPU time (low order bits)

NOTE: The ?PROC is one of the largest hogs of GSMEM. ?PROC always needs at least PLN(o3)+.PEZ(94) words for temporary PTBLs and PEXTNs in addition to that required by other routines called by ?PROC. Also, if the PIDTB has to grow (PID#>PIDLN) then PIDLN+16 words must be allocated from GSMEM for the new table, which is then copied from the old table before the old table is returned to GSMEM.

System call trace:

1. User parameter is copied to CB stack

2. Privilege of unlimited sons checked. If user does not have the privilege, then check against maximum number of sons.

3. Check to see if callers does not want to block. If so does the caller have this privilege.

4. New process table is allocated from GSMEM. Temporary PTBL ext. is allocated from GSMEM

5. Set up initial PTBL entries. Status word contains daemon request bit and swapin in progress bit set..

6. .PR file is opened, file type is checked, access is checked

7. Block 1 (400-) of program file is read in and TCB addresses are verified

8. Buffer for block 1 (step 7) is released. FCB for save .PR file is mapped

9. SWAP space is allocated by WSWP (module SSDV3). This allocates approximately 10 blocks for system use and 4 * (# of unshared) blocks for the program

10. User name is set up

PROC

PROC2

11. Initial and default directories are set up, pathnames resolved access checks are performed

12. Process ID is determined. (If ID table needs to be expanded, a new table is allocated from GSMEM and the old table is released

13. If PID # is reserved by the connection manager, release it and try again (step 12)

14. Process name is set up by SPNAM (module PNAME) and name is stored in the PIF (Process Information File)

15. IPC spool file created and initialized. IPC message sent to GHOST

16. Searchlist set to father's searchlist

17. Pathname of the .PR file (starting from the root) is stored in the PIF

(PROC2 continued)

18. If the PROC package has an extender, process it (JSR into PROC3).

19. A quarter page is allocated for the PTBL extension and is mapped in. The temporary extender is copied to the real extender and is released. The process primary and GHOST map words are set up in the quarter page.

20. A console is assigned and the initial IPC message is sent.

PROC2

PROC3

21. If requested, pass the fathers DACL to the son.

22. Copy the new PTBL into its virtual PTBL and return the temporary one to GSMEM.

23. If process was created with block, manually set the block bit (the normal routine to do this cannot block with an outstanding system call (i.e. this one) is in progress.)

24. Reset reschedule bit so caller will run again.

25. Set up connect time/day, initial exponent and enque factor.

26. The process table is enqueued to IESWP (if swappable) or IERES (if pre-emptible or resident)

27. It is now up to COREM to handle the initial program load (using a DAEMON, getting core, etc. Off we go ... (see next page)

Time passes ... Eventually the PTBL will get control of the CPU and it initial load daemon will too. Then ... (we are now in SSOV3)

1. Set the shared area change bit.

2. Get the CCB referenced by the ?PROC call and set up some values (i.e. this is a read request, start reading from byte 0 ...)

3. Using the size of the unshared area, calculate how many blocks to read in.

4. NQ the CCB and wait for completion (NQCCB). (this actually does the read for the unshared area.

5. Set up the process stack and enter debug flag if necessary.

6. Set up the initial time slice and store it in PTBL.

7. If we need a ghost (entering into debugger or have a shared area) set up and chain to ghost init. code. If no ghost, return (and release this daemon's CB, and reschedule self to start running).

PRCER Error handling routines for PROC, PROC2, PROC3

1. Error code is saved on stack

2. If PID exists, PID is removed from ID table and from father's son list

3. If virtual PTBL extender exists, it is mapped

4. If any PTBL extender exists, then the default and working directories are released, the IPC file is cleared and released. The program CCB is released, the swap file space is deallocated, and then the extender is released.

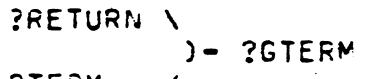
5. If the temporary process table exists, the space is returned to GSMEM

6. The error code is passed from the stack to AC2 and control is given to RETE2 in module (SCPRC)

4.5.5 Process Termination

There are basically five ways that a process can terminate.

1. Direct termination -- (self termination and forced termination by a different process)



2. Console interrupt -- (user forced by typing interrupt key)

^C ^B

^C ^E

3. Trap

Write protect I/O protect

Validity protect Indirect protect

Database Error

4. Father termination

5. Fatal process error - (While processing a system call or internal routine, AOS has taken an unrecoverable error pathe and must terminate the process (this happens most in CHAIN but can also happen in SSOV3 and SSOV6.

Code paths:

Self or forced termination

1. Save current TCB and PTBL addresses.

2. If not self termination, determine PID of target.

3. Validate PID and make sure the user has the privileges required to terminate the target.

4. Set the self term bit.

5. Save the TCB address in the PEXTN.

6. Zero offset 0 of the CB.

7. Join common code below.

Console interrupts

1. If PID = 2, ignore ^C^B and ^C^E.

2. Disable input ring buffer (no more input)

3. Set console interrupt bit in PTBL and if ^C^E set breakfile request.

4. Unblock process (if need be).

5. Reschedule process to run at PRCNI.

6. Join common code

TRAPs

1. Determine trap type
2. If system trap -- PANIC
3. Abort flag and trap bit turned on in PTBL.
4. Trap information saved in PEXTN.
5. Process is rescheduled to run at PRCNI (in PRCNG).
6. join common code

Fatal Process Errors

1. Error code is placed in CERWD of control block and in the PEXTN.
2. Fatal term bit is set.
3. Join the common code

Common code

1. Search for lowest son on process tree. As you go:
   a. Set the process terming bit (sometimes referred to as first term bit).
   b. If process is unblocked and has sons block it and set the master block bit on.
2. When lowest son is found call PBITS (in SGSUBS) which will:
   a. Set process now terminating (really terming not first term)
   b. Make sure process is in core and unblocked.
3. Start a termination daemon for this son.

Termination daemon

1. If process has any active control blocks, send until they are finished.
2. If PID = 1 (PMGR) and we are processing a trap -- PANIC

3. If trap or ^C ^E call IBRK which will create a breakfile.

   a. Generate breakfile name (use 16 words allocated from GSMEM).

   b. Delete / Create / Open the file

   c. Set up the CCB for the file (put breakfile CCB address in PTBL)

   d. ACL breakfile (ACL from PEXTN so we might have to load the PEXTN).

   e. Release breakfile name space to GSMEM.

4. Log file message is built and sent (we are now in CLNUP)

5. Release user devices (IDEF)

6. If a breakfile CCB exists (i.e. we are creating a breakfile) then do the following in this order:

   a. Write out the user unshared
   b. Write out the user shared
   c. Write out the ghost unshared
   d. Write out the ghost shared
   e. Close the breakfile

7. Send a termination message to INFOS (if this process was associated with it).

8. If RMA is terminating, set internal AOS flags to show that there is no RMA.

9. If the AP is defined (APSIZ<>-1) and this process was the current owner (APPID=current PID), then zero APPID and the users 'do not swap flag'.

10. Release user shared area.

11. Release ghost shared area.

12. Close shared area CCB (if any)

13. Close any open files (JSR to overlay RESET)

14. Release searchlist.

15. Inform the PMGR that this process is terminating.

16. Delete process IPC entries.

17. Dequeue outstanding IRECs

18. Close IPC spool file.

19. Inform the Connection manager about the termination (TBC in CONX)

20. Release all quarter pages associated with the user's CCBs.

    Remove outstanding delays and histograms from the appropriate chains.

22. Dequeue process from ELQUE. (end of CLNUP)

23. If father is not terming, and this is not the root process, send termination IPC message to father

24. Deallocate swap space.

25. Release working and initial directories.

26. Unlink process from father's son list.

27. Call RCORE which will release the primary and ghost unshared memory, maps, and extender associated with terming process.

28. If father is waiting for son termination, and father is not terming, unblock father.

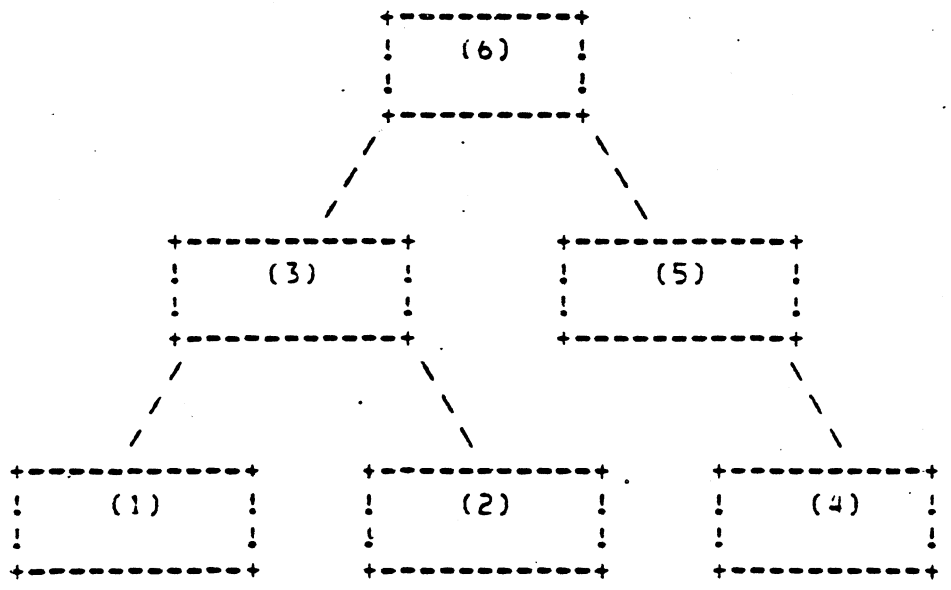
29. Remove PID from PIDTB and PIDBT.

30. Release the PTBL.

31. If father is also terminating, goto common code (this is reentrant) else we are done.

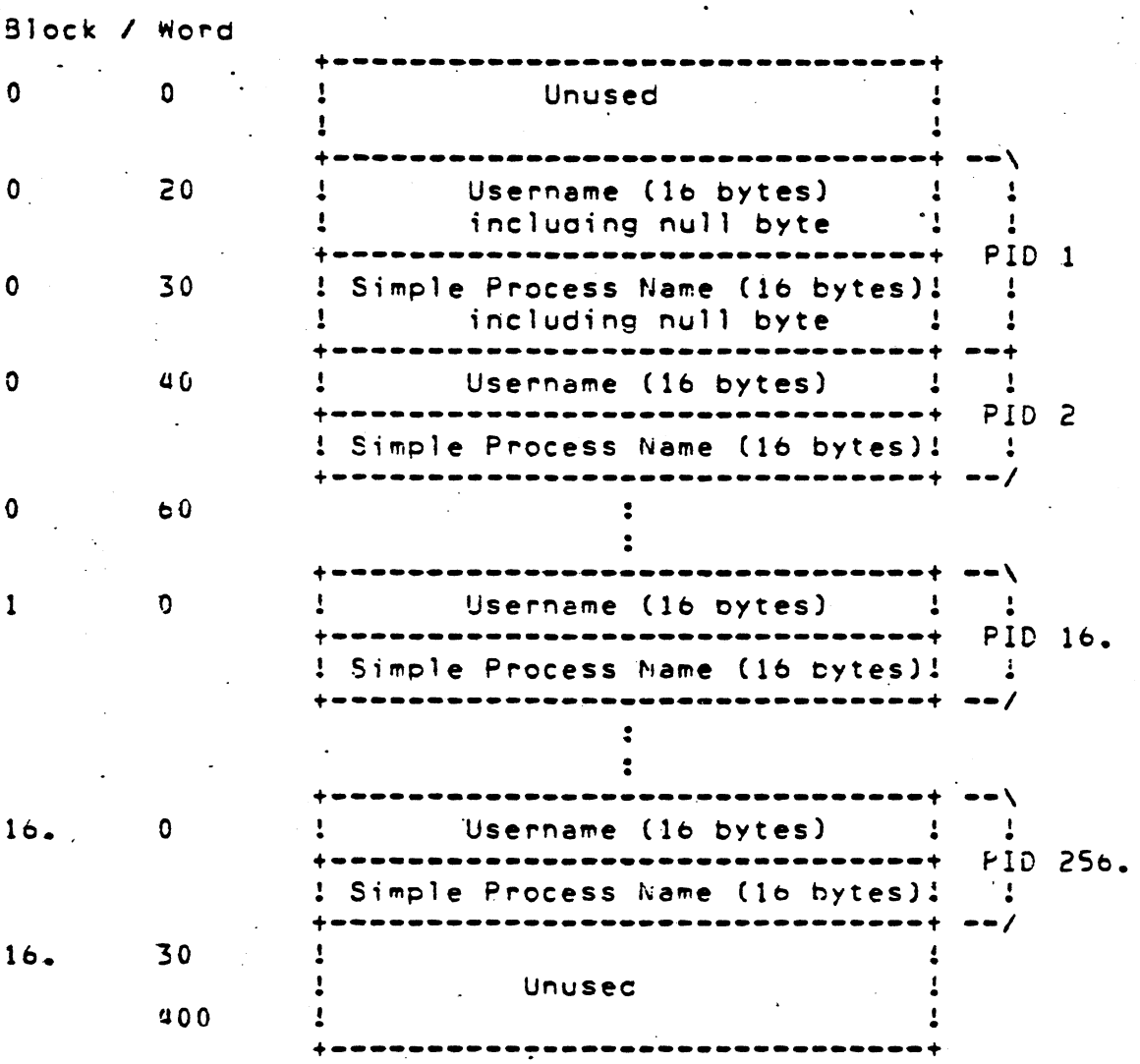
Note: A swappable process that is terminating is given a priority of 180 (highest allowable PNQF for a swappable PTBL)

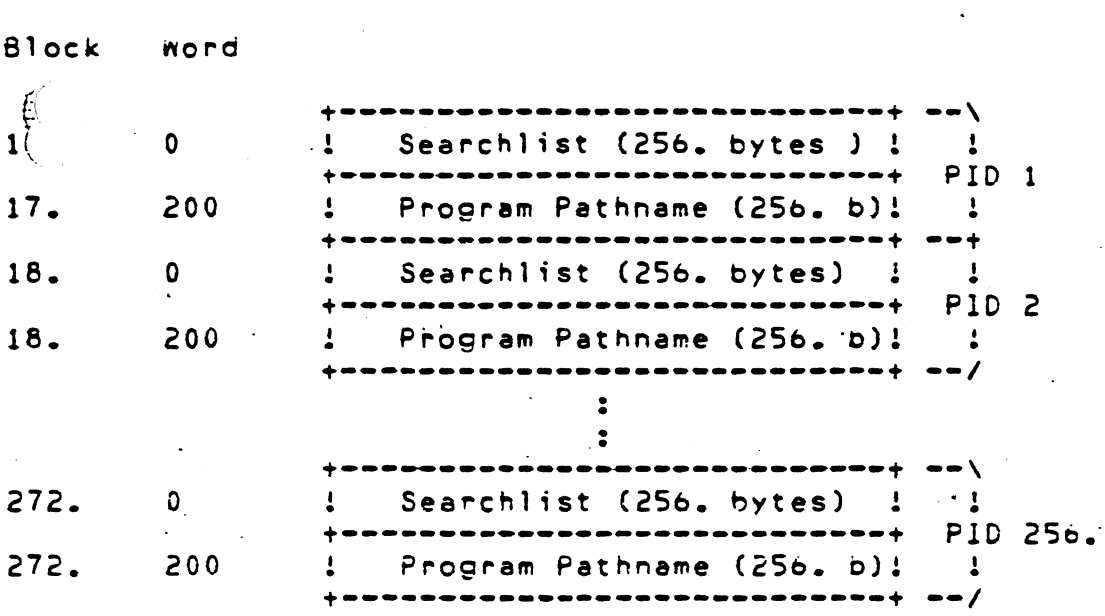
Process termination diagram -- the numbers indicate order in which the processes will terminate.



4.5.6 Process Information File (PIF)

The process information file is created at system initialization time and is continuously updated by the system. It is indexed by PID and contains information concerning the username, process name, searchlist, and program file for the user.





4.5.7 System shutdown

1. Term all processes (set term bits in all processes)
2. Pend root process until all others are gone.
3. Turn off system log.
4. Close PIF
5. Release :PER directory CCB
6. Close swap file.
7. Release all LDUs still initialized (the root [:] is last to go)
8. Tell the world and halt the processor.

4.5.8 The ?DELAY call

The ?DELAY system call pends a user task for a user-specified amount of time, without tying up system resources such as a control block. It is a direct call, i.e. does not involve a control block (chapter 6). All processes that have outstanding ?DELAY calls are linked on the delay chain, described in section 3.4.1.

The implementation of delays can be divided into five parts :

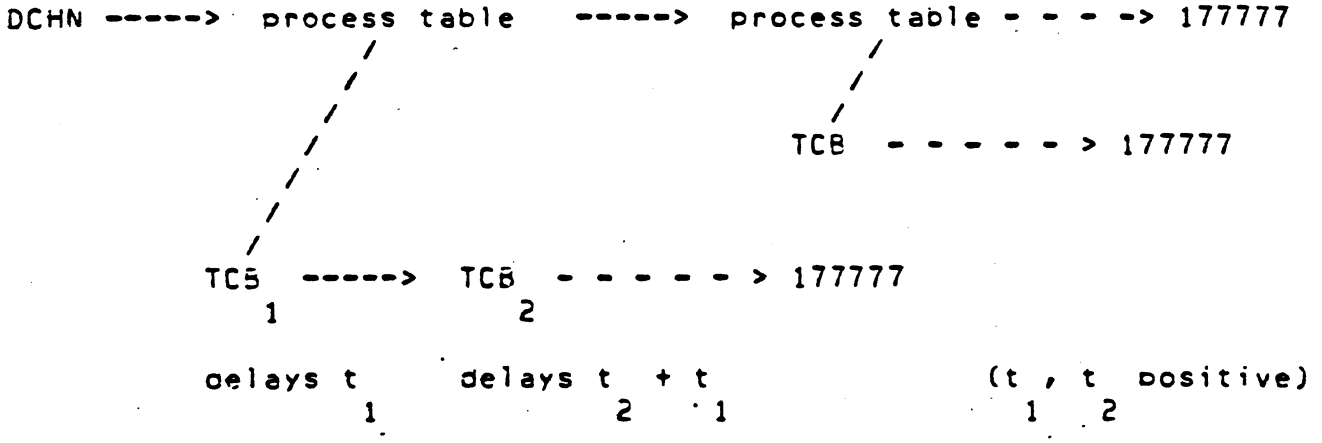
- Processing the delay call and enqueuing the request.
- Monitoring the amount of time remaining before the delay completes.
- Unpending the task when the delay completes.
- Cleaning up any outstanding delays upon process termination.
- Cleaning up any outstanding delays upon a task abort.

4.5.8.1 Processing the ?DELAY call and enqueuing the request

This part is done in the resident overlay SCMOD. When a user task issues a ?DELAY call, the process table is enqueued to the head of the delay chain, unless the process is already on this queue due to another task being still pended on a ?DELAY. The time to delay is converted to real time clock ticks and interrupts are disabled so that a clock interrupt can't come in and cause the delay chain to change from underneath.

If there are other tasks making a ?DELAY call, the TCB issuing the present call is linked to the others through offset ?TSYS so that TCB's for this process are ordered by the amount of time left to delay. If two tasks delay for the same amount of time, the link word of the first task will have bit zero set (1B0). The list is terminated by a minus one. Then the process table on the delay chain is updated, with, offsets PDINL and PDINH containing the number of real-time clock ticks that the first task is to delay, and offset PDFR of the extender the TCB address of this first task.

Hence delaying tasks TCB's are ordered on a per process basis :



4.5.8.2 Monitoring the time remaining and unpending the task

This is done at interrupt level by the real time clock interrupt service routine RTCIS in DVOV6. The delay chain is scanned, and for every process on it the process table is retrieved and the time remaining to delay (offsets PDINL/PDINH) counted down. If it does not go to zero there is nothing else to do.

Since most of the information required to unpend a task is in the TCB, which is part of the user's address space, a task can be unpended only when the process is eligible.

When the delay for the first task on the process TCB chain of delays expires and the process is eligible and not blocked, the extender is mapped in order to identify which task to unpend (offset PDFR). The user page zero is mapped to slot 76000, the task unpended, the chain rearranged and the new (next task's) delay information, if any, is stored in the process table. The process is rescheduled via a call to IWKUP, and if this was the last task on its delay chain, the process table is dequeued from DCHN.

If the process is eligible but blocked, an unblock request is posted to the core manager and the first delaying task unpended as above, by mapping the extender and the user page zero, updating the delay offsets, readying the process and dequeuing the process table from the delay chain when appropriate.

If the process is ineligible, the flag bit PFDUP in its process table ("Unpend TCB at head of delay chain") is set for the swap in and unblock code. Nothing else can be done at this point. The core manager recognizes this bit when a swap in or unblock takes place and resets the time to delay to one clock tick. This forces the interrupt world to reprocess the delay completion at the next real time clock interrupt.

4.5.8.3 Cleaning up outstanding delays

Cleaning up outstanding delays includes process termination (in overlay CLNUP) and task abort (in SSUV6).

If a process is on the delay chain, i.e. has at least one task with an outstanding delay when it terminates, the process is dequeued from the delay chain.

If the target task of a task abort (?TABT) is pended due to a ?DELAY, the task is dequeued from that process's TCB delay chain. If it is the only task with an outstanding delay for this process, the process is dequeued from the delay chain. Otherwise, the delay information in the process table and the process table extender is made current with the values from the new first task on queue.

4.5.9 The System Memory Key MKEY

MKEY is a location defined in the system upper page zero to keep track of the significant changes in available system memory (GSMEM). When a process has to wait for memory, the current value of MKEY is stored into its process table in offset PMKEY. Subsequent comparisons of PMKEY to MKEY allow the scheduler to decide if the memory situation is now such that the process could in fact run.

Besides STABLE, MKEY is referenced by the following modules :

MEMRY - Routine GSENQ ISZ'es MKEY when adding an area of GSMEM space to a significant chain, i.e. when the element is added to an empty chain and all chains of larger size elements are also empty.

SCHED - Routine MKEL will not move the candidate to ELQUE if he is waiting for memory (bit PSMWT in status word) and his PMKEY is equal to the system's MKEY. This unless one of his higher priority bits PSBRK, PSUNP, PSBAG, PSBLK, PSDP is set.

- Routine PMWT, dispatched on by the process scheduled start up code (PENTR) when bit PSMWT is set, checks if PMKEY is different from MKEY. If not, we will unbind the process unless, of course, the "Do not make virtual" bit PFNVT is set...

SCPRC - Routine TERTN (bad return from control block processing) will restart the TCB request when we cannot get the memory resource to process it. Before releasing the control block the process is marked by setting bit PSMWT, so that it will not run before MKEY changes.

CMOV1 - Routine FIT tries to fit in core the best candidate. If this process was waiting for GSMEM space (bit PSMWT set), we compare its memory key at offset PMKEY to MKEY. If different we can run. Otherwise we try and preempt for one page to be added to GSMEM.

SWAPM - After preempting a process we ISZ MKEY in case a process in core was waiting for this preempt. MKEY will be compared to the value when the wait occurred kept in PMKEY, to prevent constant retrial of the memory pick up. We give it another try only when MKEY has grown.

4.5.10 MODBITS support

A modbits system is defined as an M600 with the page modified hardware enabled. In this mode, the hardware will set a bit in a table everytime a page is modified in a map. Using various I/O instructions, AOS can request the bit specific for a given page. Since the user always runs on the A map, AOS will only look at the A map page modified bits.

A bit in the CME (CMCPG) is set aside to indicate that the page associated with the CME has been modified. When AOS goes to flush a shared page, it will look at this bit, and if it is set, will flush the page. Therefore, it is necessary that the bit be set before calling the flush routine for any pages that have been modified.

The page modified hardware will automatically keep track of page modifications in the A map for a given set of map words. Whenever we modify the contents of the A map, we must read out the page modified registers and put the information into the users PTBL (at offsets PMOD1 and PMOD2 for the user or PGMD1 and PGMD2 for the ghost). When a user then flushes the page, we look at the PMOD value for the page, and if it is set, we mark the page's CME as hardware modified.

It is also necessary to mark any page AOS accesses as modified. Because AOS runs on the b map, the page modified hardware will not detect changes in a user page made by AOS. Therefore we will mark the appropriate CME as modified.

Finally, it is necessary to mark the page as modified when we issue a ?RDB to that page. This is because the page will be modified through the datachannel map, and the A map's page modified registers will not reflect the change.

THIS PAGE ORIGINALLY WAS LEFT BLANK

CHAPTER 5 - DISKWORLD
(update for AOS Rev. 3.11)

The disk world of AOS, for the purpose of this manual, will be broken into the following major areas.

5.1 The physical disk, will describe the databases maintained on the disks themselves, ie. the FIB, DIB, etc. and the file information units (FNB, FAC ...).

5.2 Internal databases - those kept in memory ie. FCB, LCB, UDB etc.

5.3 I/O processing - tracing the paths of the major disk world routines.

5.4 traces a number of AOS system calls through the disk world.

5.5 a list of page zero locations relevant to the disk world.

5.1 The Physical Disk

5.1.1 Physical Layout

Physical disk units (PU)

The fundamental element of a disk unit is the disk block or sector. AOS addresses theses blocks sequentially, so that a unit with N blocks has an addressing range of 0 through N-1. These sequential addresses must be broken down into head, sector, and cylinder addresses in order for the disk unit to access the desired blocks. This translation is performed by the disk drivers.

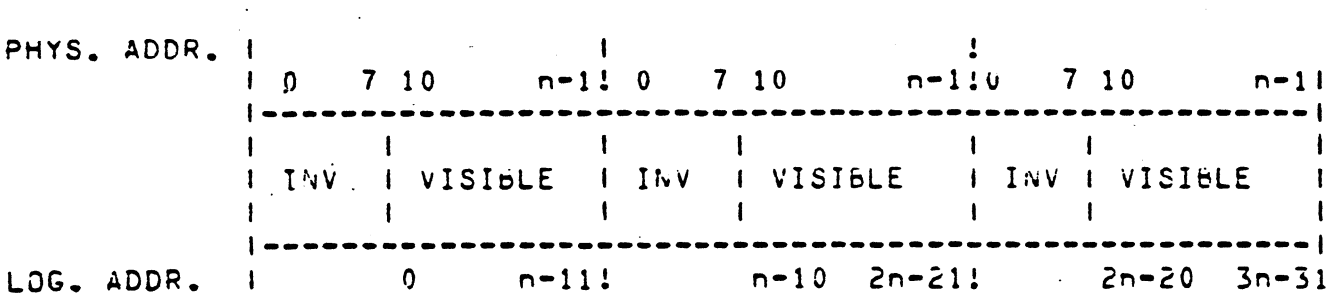
Logical Disks (LDU)

A logical disk is an association of physical disks which are made to appear as a single large disk. The purpose for this is to allow a more extended addressing space. A LDU can be composed of from 1 to 8 disk units of any mixture of AOS disk types.

In order to identify the structure of a logical disk, AOS requires a certain amount of each disk's address space to be reserved for the system's use. This reserved space is invisible to the user and therefore does not have an AOS logical disk address.

A logical disk has a number of logical blocks equal to the sum of the numbers of physical blocks on all the disks in the LDU minus the total invisible space on all those disks. Logical disk addresses must be broken down into physical disk unit and the physical disk address on that unit. This translation is done by the disk block I/O routines.

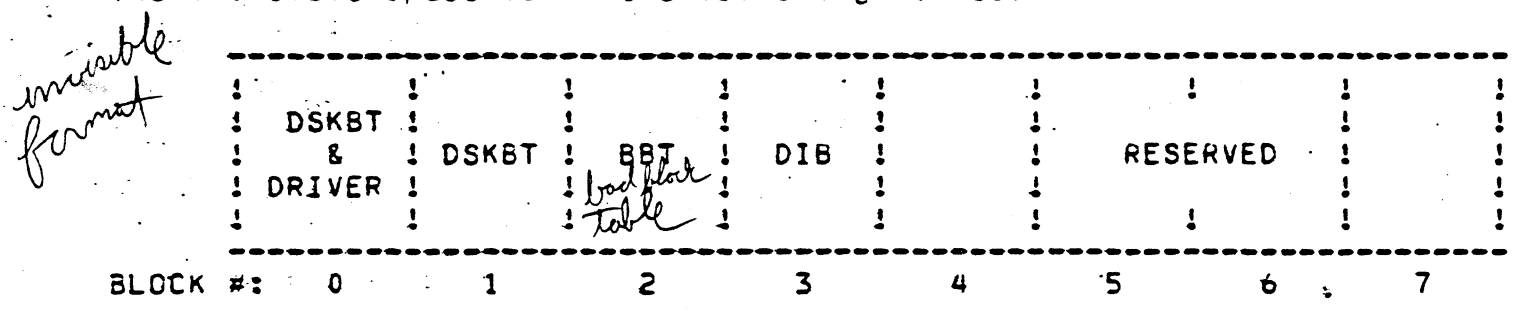
The following example compares physical and logical addressing of disks:



AOS disk format

As mentioned before, each PU in a LDU is divided into two main areas. These are the INVISIBLE SPACE which is the first 8. blocks and contain information needed by the system and the VISIBLE space which is the remainder of the disk and contains both user and system data.

The invisible space is in the following format:



DSKBT and DRIVER:

This contains the information read in by the program load switch This contains a disk driver and the logic to read in block 1 (DSKBT).

DSKBT:

Block 1 contains the code needed to read in the DIB (block 3) and fetches the location of the system bootstrap area. It then reads in the system bootstrap and transfers to the routine.

BBT:

This block contains the AOS bad block table for the PU. The table is set up as follows:

symbol	word	function	
		==========	The symbols are defined in PARFS.
BBNBB:	0:	# of bad blocks	
BBRAH:	1:	--REMAP location --	The REMAP location is the area in which bad blocks will be remapped to.
BBRAL:	2:		
BBRAS:	3:	REMAP area size	
BBB5D:	4:	-Bad block addr #1-	Bad block addr n is the address of the nth bad block.
	5:		
	2n+2:	-Bad block addr #n-	All disk addresses are two words long.
	2n+3:		

DIB (Disk Information Block)

The DIB's primary purpose is to link the units of a LDU together. At logical disk initialization time, considerable checking is done to ensure that the specified disks form a complete logical disk. In addition the DIB contains unit sizing information and pointers to system data bases.

The DIB is defined in PARFS starting at the symbol IBREV.

Briefly, the DIB contains the following information:

- Disk Format Revision number
- Per unit status flags
- LDU unique I.D.
- Sequence number of this PU in LDU
- # of physical units in LDU
- # of heads, # of sectors/track, # of cylinders on the PU
- # of visible disk blocks on the PU
- Physical disk address of the BBT (always 2 for now).

If the unit is the first of a LDU the DIB also contains the following information:

- Per LDU status flags
- Logical disk address of LDU name and Access control blocks
- Logical disk address of bit map.
- System bootstrap disk address and length
- Overlay area disk address and length
- Installed system pointer
- LDU current and maximum sizes
- "Funny FIB" of root directory (FIB = File Information Block)

root directory

VISIBLE SPACE

Several of the AOS system data bases are in the visible portion of the disk. These locations are allocated by either the formatter (DFMTR) or the installer (INSTL).

BIT MAP -- indicates which disk blocks have or have not been allocated (1 bit per block -- set if in use). There is one bit map per LDU and the location is setup by DFMTR

REMAP AREA -- is the area to which bad disk blocks are remapped. There is a remap area on each PU, the location of which is setup by DFMTR.

BOOTSTRAP AREA -- is the area which contains the code for SYSBOOT.

OVERLAY AREA -- contains AOS system overlays and is setup by DFMTR.

INSTALLED SYSTEM -- is a file that lives in the root (:) but has no name. The file is created by INSTL

5.1.2 Files in the AOS disk world.

Index blocks

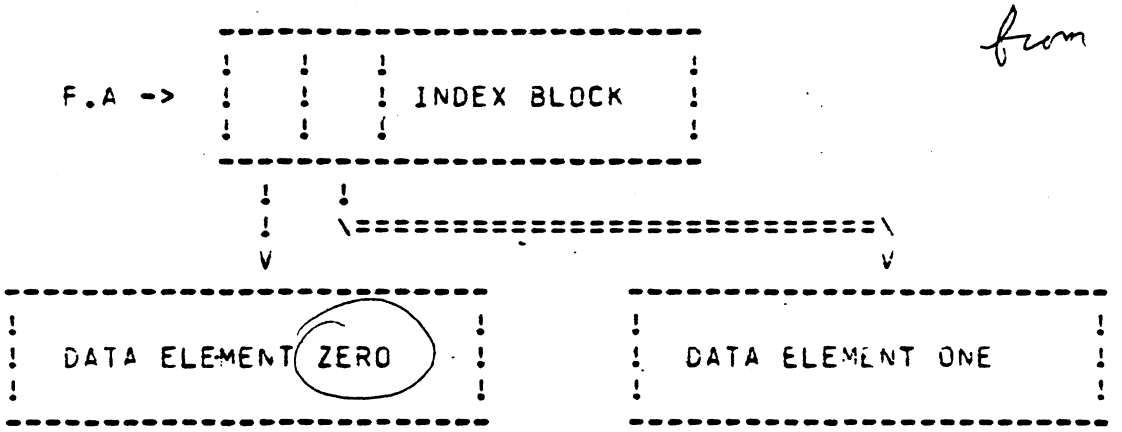
The following diagrams trace the growth of the index tree for an AOS file. F.A indicates the 'first address' or where AOS looks at when looking for the file. Each index block contains 128 double word pointers.

F.A.

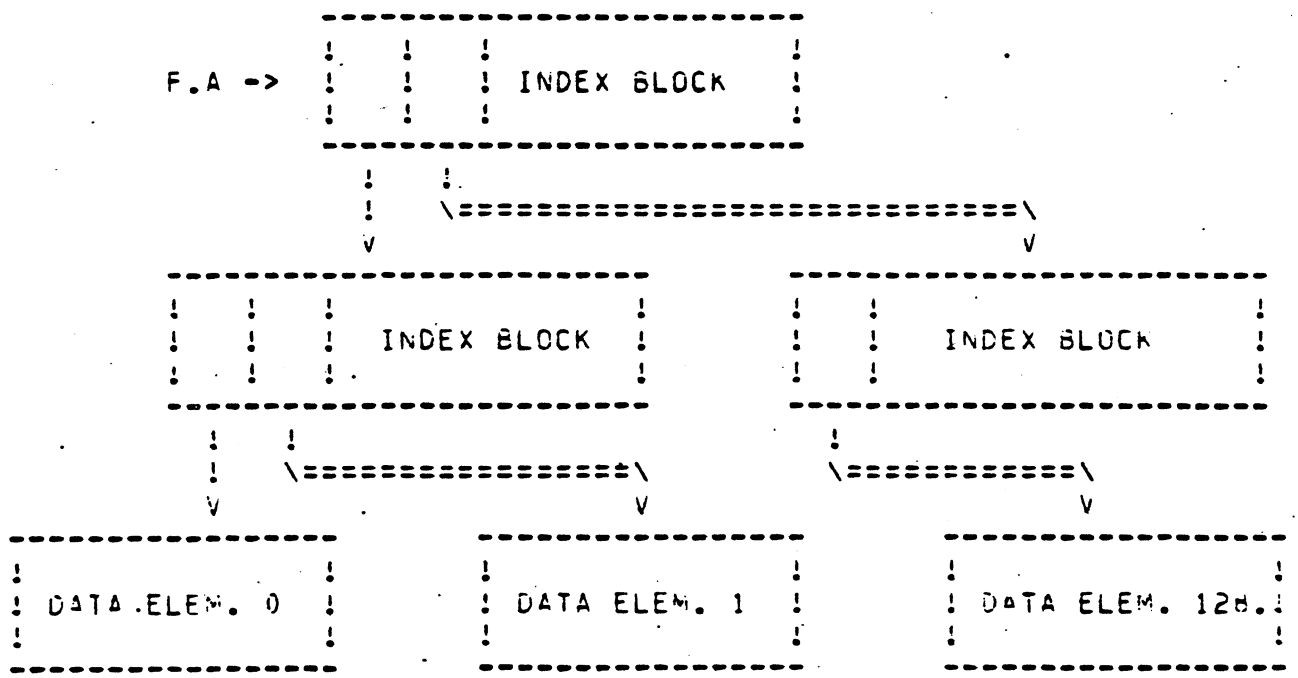
After data element zero is written:

data elements are numbered from 0

After data element one is written:



After data element 128. is written:



5.1.3 AOS disk directories

Building Blocks

The file (standard AOS file, element size of 1) is arranged in 256 word Directory Data Blocks (DDB) which consist of various Directory Data Elements (DDE). Consecutive DDEs are not chained together. The DDBs can be chained together using relative pointers that consist of single precision relative block numbers within the directory. The first eight words of each DDB is a header of the following format.

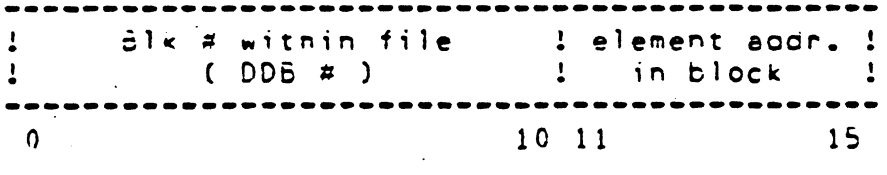
Word	Description
0	Forward Link (to other DDBs)
1	Backward Link (to other DDBs)
2-7	0

Subsequent words are allocated in the eight word elements to form the DDEs. There are five types of DDEs. They are:

Symbol	Element type code	Description
DEFNB	1	FNB (File Name Block)
DEFAC	2	FAC (File Access Control)
DEFIB	3	FIB (File Information Block)
DEFLB	4	FLB (File Link Block)
DEFUD	5	FUD (File User Data)
DEFRE	0	(Not really a type of DDE, 0 indicates free element)

The main database is the FIB. It contains or points to all vital information about a file. The FIB will point to the first FNB, the FAC and, if defined, the FLB and the FUD. All of these databases in turn point back to the FIB.

The pointers between the FIB and other Directory Data Elements are called IDPs (IntraDirectory Pointers). This format is as follows:



The offset 0 (DETAS) of any of the DDEs is special. The left byte contains the element type (0-5) and the right byte contains the size in words. Note that the size of a free element (type 0) is 0 (i.e. offset 0 is 0).

The remainder of the offsets vary from DDEs. These are defined on the following pages.

File Information Block (FIB)

Symbol	Offset	Description
FINLP	1	Pointer to first FNB (IDP)
FIACL	2	Pointer to FAC (IDP)
FILBP	2	Pointer to FLB (link only) (IDP)
FIUID	3	Unique ID
FITCH	4	File creation time (hi)
FITCL	5	File creation time (lo)
FISTS	6	File status
FITYP	7	File type (RH) and format (LH)
FICPS	10	File control parameters
FIHFS	10	Hash frame size (directories)
FIDCU	10	Device code (left), Unit number (right) Unit type only
FIHID=FIHFS	10	Host ID - Network type files
SFIBL=FICPS-DETAS+1=11		Short FIB length
FIB extension for data files and directories:		
FIFW1	11	Extension for EOF in future
FIFW2	12	" " "
FIEFH	13	Last logical byte (EOF) (hi)
FIEFL	14	Last logical byte (EOF) (lo)
FIDFH	15	Data element size (hi)
FIDFL	16	Data element size (lo)
FIFAH	17	First logical address (hi)
FIFAL	20	First logical address (lo)
FIIDX	21	Current index levels (left) Maximum index levels (right)
FIIDR	22	Count of inferior directories
FIFUD	23	Pointer to FUD
FITAH	24	Time last accessed (high)
FITAL	25	Time last accessed (low)
FITMH	26	Time last modified (hi)
FITML	27	Time last modified (lo)
FIFW3	30	Extension for FCB address
FIFCB	31	Virtual FCB address or zero
FIBLT=FIFCB-DETAS+1=32		Full FIB length
FCOML=FIIDR-FISTS+1=15		Length of common data between FIB and FCB
FIB extension for Control Point Directories:		
FICSH	32	Current size (high)
FICSL	33	Current size (low)
FIMSH	34	Max size (high)
FIMSL	35	Max size (low)
LFIBL=FIMSL-DETAS+1=36		Long FIB length

Offsets 6 - 23 (enclosed by the cashed lines) are common to FIBs, DIDs and FCBs.

Symbol	Offset	Description
File Name Block (FNB)		
FNFIB	1	FIB pointer
FNNAM	2	File name offset
FNBLT=FNNAM-DETAS	2	FNB header length
Access Control Block (FAC)		
FAFIB	1	Pointer to FIB
FAACL	2	Access Control List offset
FACLT=FAACL-DETAS	2	FAC header length
File Link Block (FLB)		
FLFIB	1	Pointer to FIB
FLLCN	2	Link data offset
FLBLT=FLLCN-DETAS	2	FLB header length
File User Data block (FUD)		
FUFIB	1	FIB pointer
FUFFL	2	FUD forward link
FUFBL	3	FUD backward link
FUUDA	4	User data offset
FUDLT=FUUDA	4	FUD header length

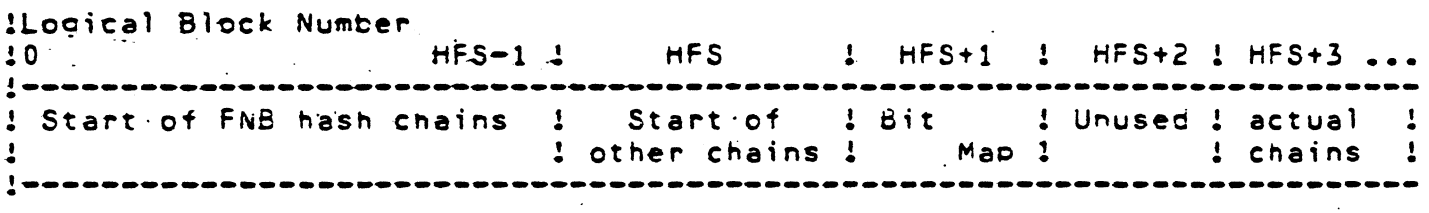
Sample FNBs (FLBs are in the same format)

DOE type length

DETAS=0	1	10 (8.)		0	1	20 (16.)	
FNFIB=1	pointer to FIB			1	pointer to FIB		
FNNAM=2	"F	"I		2	"L	"O	
3	"L	"E		3	"N	"G	
4	"N	"A		4	"_	"F	
5	"M	"E		5	"I	"L	
6	<0>	don't care		6	"E	"N	
7	don't care			7	"A	"M	
				10	"E	<0>	
				11	don't care		
					:		
				17	don't care		

Directory format:

The following is a diagram of the internal organization of a directory file:



HFS = Hash Frame Size

The first HFS blocks of a directory contain the first filenames for files that hash to the same value. Using the standard DDB links, additional DDBs are linked to these first blocks. Each DDB so linked will contain only filenames that hash to the same value. Only specific DDBs are allocated for FNBs. The easiest way to describe which uses the bit map. Every fourth word in the bit map is used to mark DDBs used exclusively for FNBs. (16 DDBs for FNBs, then 48 DDBs for other chains etc.)

The hash value is computed as follows:

1. Take the sum of the ASCII values of the characters in the name
2. Divide by the HFS (Hash frame size)
3. The remainder of step 2 becomes the hash value for the name.

Example for the filename AOS and a hash frame size of 7.

```
A = 101
O = 117
S = 123
    -----
      343 --> 343/7 = 40 with a remainder of 3.
```

Therefore the hash value is 3.

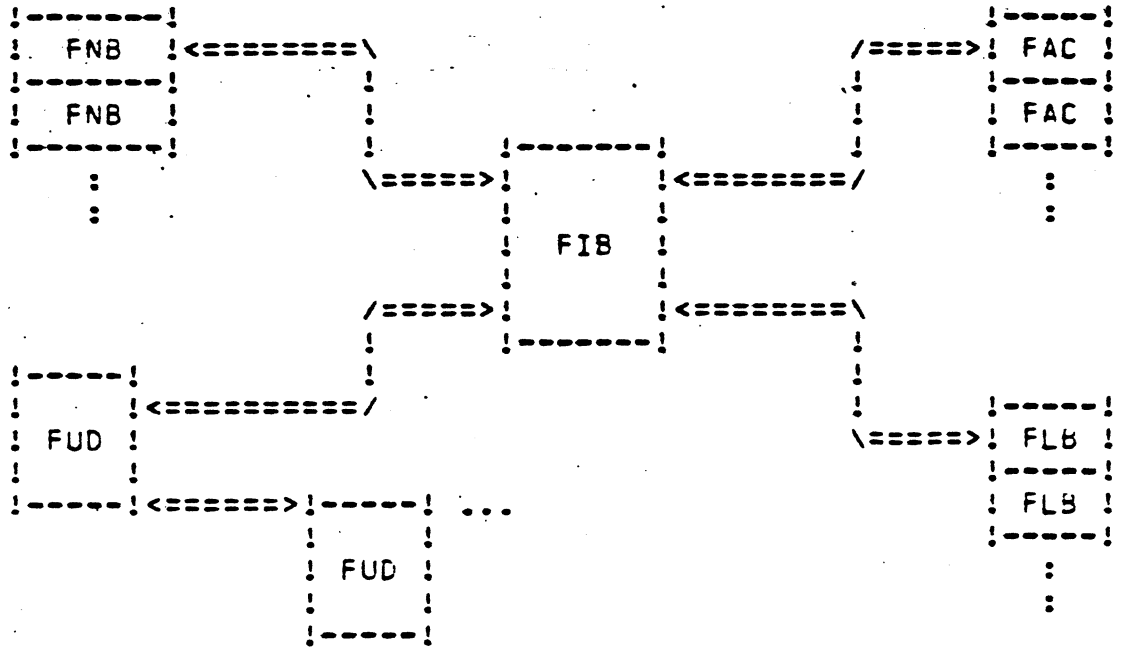
No ordering is attempted of DDBs within a chain or of FNBs in a DDB. A filename, once a hash chain has been determined, is searched for sequentially.

The next block of the directory is the anchor DDB of an unordered chain of DDBs that contains the remainder of the DDEs that make up the directory.

The directory bit map is used in allocation of blocks within a directory. It contains one bit per DDB. As mentioned above, every fourth word of the bit map is reserved for FNB allocations.

The blocks that follow the unused block (HFS+3 and on) contain the other DDBs that make up the chains.

FNBs must be in a DDB reserved for them. FACs, FIBs, FUDs, and FLBs are mixed together in the other DDBs. Pointers are bi-directional and are in IDP format.



5.2 In-core databases

The following database reside in memory. The length of time that the databases are around varies. The LDBs are around from INIT to RELEASE, the UDBs from OPEN to CLOSE, while the DCTs are permanently allocated.

LCBs and UDBs are used to convert a disk request for a logical disk into a physical request to a specific disk unit. The UDB and the DCT are used for processing the physical request.

5.2.1 Logical disk Control Block (LCB)

The LCB contains the following information:

Pointer to the chain of UDBs that make up the LCB
Cache list pointer (cache entries associated with this LCB)
Pointer to the next LCB in the system
LDU bit map lock(0=>free)
Bit map FCB real address
Bit map FCB virtual addr.
Bit map buffer address
LDU's root CCB addr. or 0
Current size (two words)
Max size (two words)

There is one LCB per logical disk that has been initialized. A LCB is LBBLT words long (20). LCBs are allocated from GSMEM and are returned when the LDU is released.

5.2.2 Unit Definition Block (UDB)

DCT address
Device unit number
Unit request list
Last logical address (two words)
UDB forward link (logical)
UDB forward link (physical)
Unit start addr (two words)
Number of blocks to move
Unit status word
Cylinder size in sectors (UDNSC*UDNHD)
Number of sectors per track
Flags (left byte) Number of heads (right byte)
Data address (two words)
DOA word (temp)
DOC word-used as running cylinder number
Error counter, flags, status, retry count
Temp block counter
Unit status (DIA or DIC only)
Bad block table pointer and remap address (two words)

Certain offsets are used by UNERR, the unit error report routines while other are used for fatal (hard) errors and in times of PANICs, etc.

Moving head disk UDB states (UDSTS)

Symbol	Bit	Description
DPIDL	0	Idle
DPSKR	1	Seek ready
DPSKP	2	Seek in prog
DPSKD	3	Seek done
DPIOR	4	I/O ready
DPIOP	5	I/O in prog
DPIOD	6	I/O done
DPRCR	7	Recal ready
DPRCP	10	Recal in prog
DPRCD	11	Recal done
DPSKE	12	Seek error
DPIOE	13	I/O error
DPFTE	14	Fatal error
DPGST	15	Get status

There is one UDB for each disk, LPT and tape unit that has been opened. (There are different symbol definitions for the tape not discussed here.) A UDB is UDSLT words long (40), allocated from GSMEM, and released when the unit is closed.

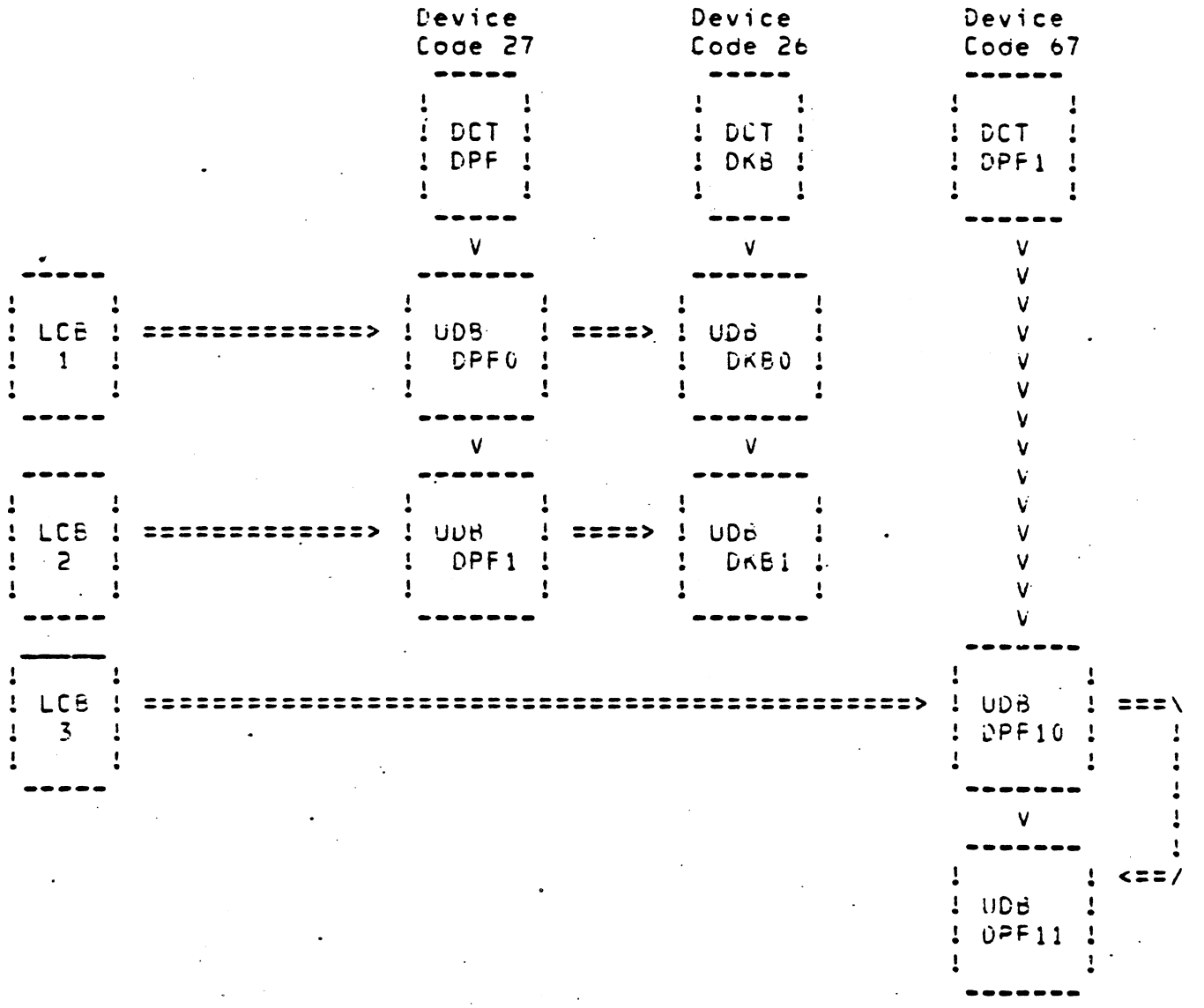
5.2.3 Device Control Tables (DCT)

The DCTs contain the following information:

Device code
Interrupt mask
Current I/O request
Current I/O request temporaries (error status ...)
Map slot assignments
Address of device specific routines (Initialization, I/O ...)

DCTs for AOS are bound into the .SY file at AOSGEN time.

The following is a diagram of how some database are connected. It assumes three LCBs, two containing a fixed head disk and 6060 (96 mb) and a third containing two 6060s.



5.2.4 File Control Block (FCB)

There is one FCB for each file opened regardless of how many users open it. FCBs are 32. words long and contain the following information:

Pointers to LCB, CPB
"Funny FIB" for the file (file type, status, EOF, element size
Hash frame size, first address, index levels)
Open count
Parent CCB pointer
File level counter
Pointers to buffer and shared page chains

FCBs are loaded from the disk FIB when the file is first opened. The use count is incremented for each user that opens a file, decremented when they close it. When the use count returns to zero, the FCB is released. FCBs are found on pages in AOS's disassociated space.

5.2.5 Channel Control Block (CCB)

There is one CCB for each channel (user or system) open. The CCBs are 16. words long and contain the following information:

Pointer to the FCB
Parent CCB pointer
Number of blocks to transfer
File data block number
User buffer address
Last block byte count
Priority
Retry count (MTA/MCA)

System CCBs are in GSMEM. User CCBs reside in hyperspace and are swapped out when the user goes. They will not swap in automatically when the user swaps in, but rather when the first system call references the CCB.

5.2.6 Enqueue blocks (NQBLK)

NQ blocks are temporary databases that contain immediately relevant information taken from a CCB. They are used as an efficiency measure to eliminate the mapping of CCBs each time they are needed. NQBLKs are allocated from GSMEM.

5.2.7 Control Point Directory Block (CPB)

There is one CPB for each Control point directory or LDU. Each is 8. words long and contain the following information:

Current size in disk blocks
Maximum size in disk blocks
Pointer to parent CPB

Every time a disk block is allocated or deallocated for a file, its parent CPB (pointed to by the FCB) is incremented or decremented. The CPB's parent CPB is also modified recursively. CPBs are allocated from GVMEM.

5.2.8 Buffer Header (BH)

Buffer headers contain information about buffer level I/O. They are enqueued off the appropriate UDB. They contain the following:

Buffer Address
Data address on disk
Number of blocks
Mapping information

Buffer headers are in GSMEM.

5.2.9 Input/Output Control Block (IOCB)

Symbol Offset Description

IOCBs contain the following information:

CCB address
Forward and backword links (for IORUN -- chain of IOCBs)
IOCB status word
Save levels (IOCB routines do not use the stack)

The following locations are used by the diskworld as a buffer header to enqueue the request (until the dashed line)

Data address
Status
Link to next BH
#blks
MAP-comes from CBMAP
unpend(post process) address
TCB address
Data address (high)
Data address (low)

Counter of #levels of indexing needed
Indexing word(high).has <0><x1>
(low). has <x2><x3>
Current #levels in file
Temp variable IOICB(=IOVAR)
Hold IOCB address through BUFIO
Q=file element #=blk#/element size IOQLO (two words)
Remainder from Q computation(two words)
Data element size (two words)
Amount of bytes transferred so far
Mapped and virtual FCB addresses
Block count
Buffer header address

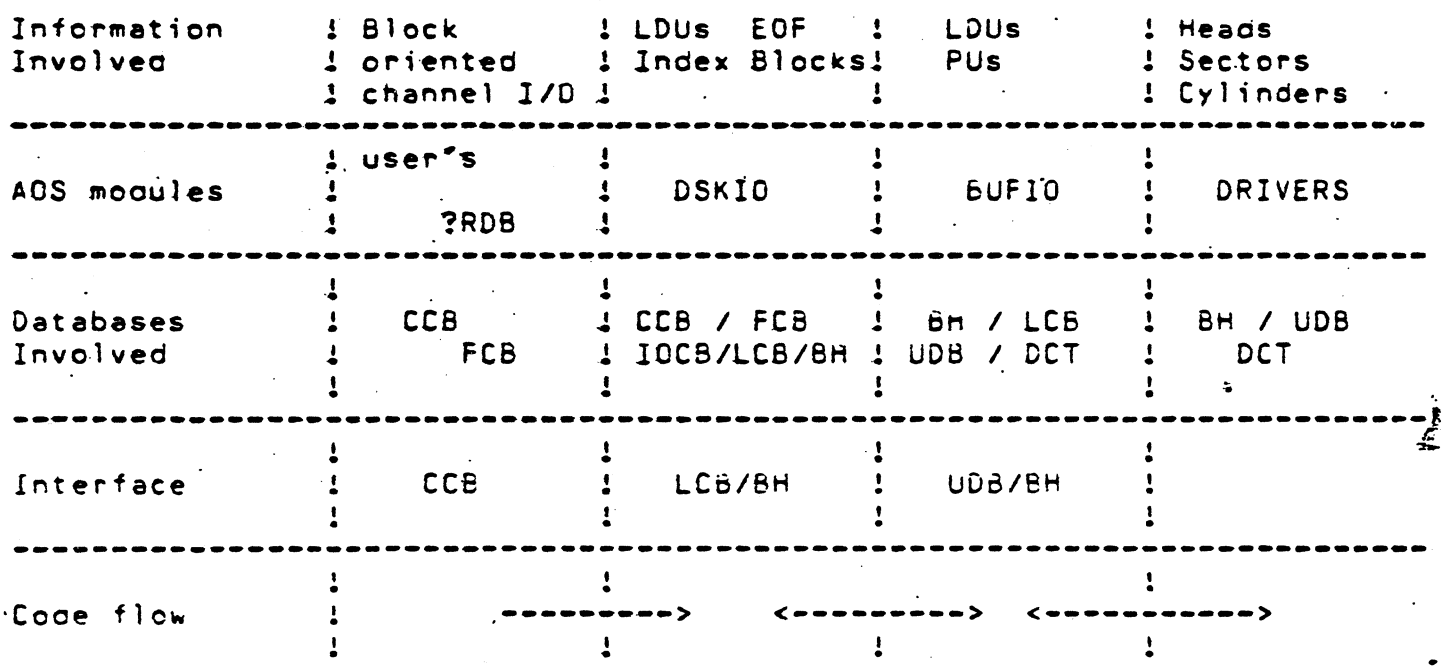
The save levels are used to hold return addresses from subroutines. IOCBs are allocated from GSMEM as they are needed (see below) and are 64. words long.

IOCB status states:

IOCB dormant
Waiting for I/O
IOCB ready
Waiting for a buffer
Waiting on the bit map
Waiting on file IOP

5.3 I/O Processing

The following diagram is an attempt to follow a ?RDB call through the various system levels.



Notes:

1. The code can bounce back and forth between DSKIO and BUFIO in the case when BUFIO returns an index block instead of a data element to DSKIO, which then must ask BUFIO for the next level.

2. The code can bounce back and forth between BUFIO and DRIVERS in the case that a request reaches the end of a PU but not the end of a LDU.

5.3.1 IOCB Processing

1. Initialized by NQCCB. (Will allocate an IOCB, place the location of RUNRD into the IOCB at offset IOSPC and flag the IOCB ready to run.)

2. RUNLCB will find the IOCB ready to run and begin execution at @IOSPC.

3. Code path will have to pend at points (waiting for disk req.). This is done with a call to WAIT, which will put the pending address in the IOCB at offset IOSPC and jump back to RUNLCB.

4. When the disk request is completed, the interrupt will wake up the IOCB by setting its ready to run flag.

5. Loop back to 2. This will continue to the IOCB processing is completed.

6. IOCB allocation:

   a. The max number of IOCBs for the system is calculated as follows:

      Max (IOCB for system) = [(# of VPs) + 2] * 1.5

   b. The min number of IOCBs is defined by SCMNI which is 2.

   c. The system will dynamically grow the pool of IOCBs varying the number between the min and max. When an IOCB is done and at least one IOCB is on the free chain, AOS will return the completed IOCB to GSMEM. (i.e. AOS attempts to keep at least one IOCB on the free chain at all times, and always SCMNI IOCBs allocated.)

   d. When attempting to allocate an IOCB, GIOCB first attempts to assign one from the free chain. If this is empty, GIOCB will attempt to grow the IOCB pool up to the max. If it can, it does, and returns the new IOCB to be associated with a CCB. If it cannot, the it takes an error return which forces the CCB to be enqueued of CCBWQ.

A word about WAIT

1. Called with a JSR WAIT.

2. Store away the return address (AC3) into offset IOSPC of the IOCB

3. Store away the status (AC0) in offset IOSTW.

4. If this is an I/O wait (AC0=-1) then check the IO in progress (IOP) flag in the BH to see if the IO is already done (if IO was done to a buffer in cache, then IO completes almost immediately). If IO is done, set the ready to run flag in the IOCB status and ISZ IOSPC (take normal return).

5. Try to run any other IOCB requests (Jump back into RUNLCB)

Note: JSR WAIT has two returns (error which is often a panic and the normal return.

The following are some of the routines used in the Disk World. Some attempt is made to document in the order in which a disk request will see the routines, but strict adherence to this policy only adds clouds to a stormy sky.

5.3.2 NQCCB (Module: DSKIO)

NQCCB enqueues CCB requests. It attempts to allocate an IOCB for the request. If successful, the IOCB is enqueued to IORUN. If not, the CCB is enqueued to CCBWQ.

1. If CCB is virtual, allocate a NQBLK with the vital info in it to minimize mapping.

2. Mask out the disk world.

3. Map in the FCB.

4. Try to allocate an IOCB. If this fails, goto step 11.

5. Store the CCB or NQBLK address in the IOCB.

6. Set the IOCB status to 'ready to run' and put location RUNRD in offset IOSPC of the IOCB. (RUNRD is the address at which IOCB processing begins).

7. Put this IOCB at the end of the list of IOCBs (IORUN).

8. Restore the mask to allow disk requests again.

9. Set the RUNLCB flag (RLCFL) to allow RUNLCB to run.

10. Return (this was called with a JSR).

11. Enqueue the CCB in priority order on the waiting list (CCBWQ), increment the waiting CCB counter (CCWC), and if necessary the max count of waiting CCBs (CCMX).

12. Restore the mask to allow the disk.

13. Return

5.3.3 RUNLCB (Module: DSKIO)

Runs all readied IOCBs for all LCBs. The routine saves map slots 60 and 62 and then calls RNST0. For the sake of this discussion RUNLCB and RNST0 will be one in the same. In rev 3.03, RUNLCB is called from the top of the scheduler (SMON1) and when a control block pends (PEND). The flag RLCFL, when 0, inhibits entry into RUNLCB and just returns to the user. RLCFL is cleared by RUNLCB and by ESD.

1. Check RLCFL to see if RUNLCB should really run (if not just RTN)
2. Save the 60 and 62 map slots (and JSR/SAVE to RNST0).
3. Clear RLCFL (RUNLCB enter flag).
4. Find a ready IOCB. If none exist, retore the o0 and 62 map slots (by RTN to RUNLCB). Determination is done by examining the IOCB running list (IORUN) to see if any IOCBs are ready.
5. Save the IOCB address in IOCBP (IOCBP always contains the address of the currently executing IOCB).
6. Get the PTBL of the process making the request (from the NQBLK or CCB) and map in the PEXTN.
7. If we have a NQBLK instead of a CCB, convert the NQBLK into a CCB by mapping the CCB into the 60k slot.
8. Put the mapped CCB address and the mapped FCB address into the IOCB.
9. Save the LCB associated with this IOCB in LCBP.
10. Execute the code specified by the IOCB. (JUMP @IOSPC).

5.3.4 RUNRD (Module: DSKIO)

All individual IOCB code paths start here. This address is loaded into offset IOSPC of the IOCB when it is initialized. RUNLCB jumps through this offset which always contains the IOCB restart address.

Upon starting:

AC2, AC3 = IOCB address
The FCB is mapped into the o2k slot.
The LCB is in LCBP (put there by RUNLCB)
The IOCB is in IOCBP (put there by RUNLCB)

All values stored as temporaries are stored in the IOCB.

1. Get the PID from the CCB.

2. Adjust if a GHOST request. (use the PGMAP offset instead of PAMAP)

3. Store the following in the IOCB:

a. adjusted PTBL (correct MAP)	IOMAP
b. current # of index levels	IONLV
c. data element size	IODEH/IODEL

4. If a double precision element size is specified (only disk units opened as files can have this) see if the data element size is less than the starting data block number. If it is, then this is an error for disk units. Store EOF error in the IOCB and jump to step 14. If data element size is not less than the starting data block number, the file element number (IOQHI/IOQLO) is set to zero, and the offset into the element is put into the IOCB (IOREH/IOREM). Goto 7.

5. Divide the block # by the element size yielding Q, the element # in the file. A remainder indicates that the request is in the middle of a data element.

6. Store Q into the IOCB (IOQHI/IOQLO). Do the same with the remainder (IOREM/IOREH) (IOREH=0).

7. Wait for any IO in progress on the file to finish (JSR WAIT).

8. If this is a large element size (hi word<>0) move the number of blocks required from the CCB to the IOCB (IOBNL), zero the # of blocks required in the CCB, and goto step 11. (PU is a file)

9. If the request will scan across data elements, move the number of blocks to read in this data element into the IOCB, decrement the total number of blocks to read, and goto step 11.

10. Store the # of blocks to do in the IOCB (IONBL). Zero the # of blocks to do in the CCB. (Request will not span another data ele.)

11. Dispatch (JSR) to the appropriate READ/WRITE/DELETE ... code. (The command code is in the CCB (CBSTS)). See below

12. Upon returning ...

    a. If an error has occured, goto step 14.
    b. If system buffer read, goto step 14.
    c. Look at the total number of blocks to read (stored in the CCB. If this is zero, we are done ... goto step 14.

13. Increment Q(data element number) and jump to step 8.

14. We are done. Return an error code to the CCB.

15. Call CCB unpend processor. (@CBUPD in CCB) (Stored by NQCCB) If an error occurs in processing the CCB, the unpend processor for the user request will take the error code found in offset 1 of the CCB and put it in the user's task's ACO.

16. Wake up anybody waiting for the file.

17. If any CCBs or NQBLKs are waiting for an IOCB, get the first on the list, and assign this IOCB to that request (Init IOSPC to RUNRD, set the 'ready to run' bit ...). Make next CCB or NQBLK on the list first, and jump back into RUNLCB (which will eventually run the new request.)

18. Unlink this IOCB from the running list.

19. Return this IOCB to the free IOCB pool. (JSR RIOCB)

20. Jump back into RUNLCB to run any other IOCBs.

The following routines are called from the IO processing routines documented below:

INDEX - Loops through the index levels. It takes the normal return when it reaches the data level. It take the error return if it reaches a hole in the index.

INDEX calls LBKLN to get an index block. LBKLN determines if the block is still in core (system buffer). If it is, it waits for the IO to the buffer to complete and returns. If the block is not in core, LBKLN assigns a buffer for the block, NQs the buffer header, waits for the IO to finish, and returns. If no buffer is available it waits for one.

GROW - Allocates space on the disk for a file. Allocates necessary index blocks until lowest level, and then allocates a data element. Calls NQBLK (allocate 1 blk) and NQCBK (allocates n cont. elements).

GROFL - Decides if the number of index levels must grow. If so, the address of the new index block is put in the files first address, and the old first address is put in offset zero of the new index block.

CKEOF - Checks for the EOF condition. On writes, shared reads and system buffer reads, the EOF is extended. On normal reads, the number of blocks to read is decremented, and the EOF flag is set.

The following routines are dispatched to by RUNRD (step 11)

RDDEL - Reads one or part of one data element

1. Check EOF condition.
2. Check number of blocks to read. If zero, just return.
3. If shared read, jump to SHRD.
4. Compute indexing offsets (JSR PARSQ)
5. If the file first address is 0, then goto step 13.
6. Loop through the index levels (JSR INDEX)
7. If INDEX took the error return, we tried to read a hole.. Jump to step 12.
8. Set up the buffer header offsets of the IOCB.
9. NQ the buffer header (JSR NQBHR in BUFIO)
10. Wait for the IO to complete
11. Return (Jump back into RUNRD at step 12).
12. Release BH of last index block (residue from INDEX)
13. Clear core indicated by the TCB using resident overlay IMCLR. An error from the overlay will cause a PANIC 14033. (At the point at which we came to step 13, we either tried to read an empty file or attempted to read in a hole, either of these operations will return zeroes to the user.
14. Return (Jump back to RUNRD at step 12).

SHRD - Shared read

1. If we are reading in a hole, or past the end of file, we must allocate new elements and their associated indices so that anybody else requesting the read will get the same information.
2. Fill the new areas allocated with zeros.
3. Read in the data element (jump to step 8 of RDDEL)

WRDEL - Write a data element

1. See if file must grow, and if so do it.
2. Set the modified and the flushed bits in the file's FCB
3. Zero the part of the data element not being written to
4. Set up the BH offsets in the IOCB
5. Enqueue the buffer header
6. Wait for the IO to complete. If it fails goto back to RUNRD at error return.
7. Update EOF if necessary
8. Return to RUNRD at normal return

DELFIL - Delete file's space from disk.

DELFIL steps through indices deleting blocks with calls to DEBLK (delete single block) or DEBKS (delete n cont. blocks)

This ends the documentation on RUNLD.

5.3.5 NQBHR (Module: SUFIU)

NQBHR enqueues a buffer header to the UDBs. It takes as input the unpend address, the BH and the LCB. (NQBH1 takes as input the unpend address, the BH and the UDB)

1. Count the calls to NQBHR (ISZ NQBC)
2. Save unpend address in the buffer header.
3. Set IO in progress in BH. If already set, PANIC 14050.
4. Translate BH,LCB into a UDB location (JSR GUNIT)
5. Mask out interrupts
6. If nothing is on this unit's request list, enqueue this request, and start the device (JSR STUNT)

7. If something was already on the unit's request list, just enqueue the buffer header.

8. Restore the interrupt mask

9. RTN with AC0=UDB address

NQBH1 (Module: BUFIO)

1-3. Same as for NQBHR

4. This step is not necessary since we are already supplying the UDB address.

5-10. Same as for NQBHR

GUNIT (Module: BUFIO)

GUNIT translates the BH,LCB address given into a UDB address. This is done by comparing the requested address to the high address on each PU in a LDU.

5.3.6 IODON (Module: BUFIO)

This is called from the interrupt service routines.

1. If any error flags are set, pass the error to the UDB for UNERR to report.

2. If we were processing a system buffer, update the cache and goto step 4. (Cache updates are performed by CPOST in CACHE)

3. If there are more blocks to transfer, check to see if this is an LDU or single PU. If this is the last unit in a LDU or a single PU signal an error (if PU signal EOF -- if LDU signal PANIC 14052). If it is not the last unit of a LDU, enqueue the request to the next UDB in the LDU and start the transfer.

4. Call the post processor for the request.

5. If there are anymore request on this UDB, start the unit.

6. Return

5.3.7 CACHE buffering

The cache in AOS only caches system buffers. These are primarily used for index blocks and bit maps. Before rev 3.04, cache requests were handled at the NUBHR level. In 3.11 the cache check call was put at a higher level. It is the responsiblity of each routine that will use a system buffer to first check if the block is in the cache. This new method means that an IOCB will not be tied up by a request that is in the cache, the cache code will not have to mask out the disk world, and that we do not have to tranverse the index blocks in order to find a data block in the cache.

There are three basic routines involved in reading the cache. The first, CABLN will look for a cache entry given the block # and FCB address. The other two CAM<BL,LA> will look for a cache entry given the <block #, logical address> and if found, will move the data into a system buffer.

The format of the cache descriptors is:

```
                +------------------------+
   CALFL = 0    !    LRU foward link     !
                +------------------------+
   CALBL = 1    !    LRU back link       !
                +------------------------+
   CABUF = 2    !    Cache buffer pnt    ! (1b0->LCB chain;0b0->FCB chain)
                +------------------------+
   CADFL = 3    !  FCB/LCB forward link  !
                +------------------------+
   CADBL = 4    !  FCB/LCB back link     !
                +------------------------+
   CADBN = 5    !    Data block number   !
                +------------------------+
   CALAH = 6    !  Logical address (hi)  !
                +------------------------+
   CALAL = 7    !  Logical address (low) !
                +------------------------+
```

Since the cache descriptor is 8 words long, and there is a maximum of 128 cache entries, the entries will all fit on one page.

The basic cache scheme is as follows:

The FCB of each open file has an offset that points to the chain of cache descriptors for that file. Each LCB has a chain of cache descriptors for cache entries of files that have been closed. In addition, there is a LRU chain on which all cache descriptors can be found. When the system needs to reference a block in a file, we scan down the FCB chain for that file. If we find it, we BLM the contents into the system buffer, and destroy the cache entry. Therefore, and unlike before, will either be in a system buffer or on a cache chain. Data enters the cache when the buffer containing the data is needed for some other function. The ASBUF routine will assign the buffer to the requesting routine, and move the old data into a cache entry.

5.3.8 Disk drivers

Disk drivers are the base level support of disk in AOS. There are five major modules for each driver. Each DCT will point to the relevant driver modules.

Routine	Function	DCT offset
a. Interrupt Service	- device interrupts, restart, errors	DCINS
b. Start Up	- start current request, overlapped seeks	DCSTR
c. Set Up	- setup next request, calculate seeks. if idle device, it will call startup	DCSUP
d. Initialization	- Init UDB (# sectors, heads, etc -> UDB)	DCTIU
e. Enqueue	- Enqueue new request to UDB. (DKBNQ enqueues in FIFO order DKANQ attempts optimization)	DCENQ

The 6060/6061/6067 and 6063 disk drivers is virtual. Tne locations pointed to by the offsets in the DCTs are anchors to the appropriate resident overlay.

All other drivers are in AOS's address space.

For quick reference, the following demonstrates the relationships between device name and modules that handle that device.

Device Name	a	b	c	d	e
DKB	DKBIS	DKBST	DKBSU	DKBIU	DKBNQ
DPD	DPAIS	DPDST	DPDSU	DPDIU	DPANQ
DPE	DPEIS	DPAST	DPASU	DPEIU	DPANQ
DPF	DPFIS	DPFST	DPFSU	DPFIU	DPANQ
DPG	DPAIS	DPDST	DPDSU	DPGIU	DPANQ
DPI	DPAIS	DPDST	DPDSU	DPIIU	DPANQ

Note that many routines are shared. a-e refer to functions above ie a=interrupt service, b= startup etc.

5.4 System call traces

5.4.1 ?CREATE (Overlays: CREATE/SOV13)

1. Get the user packet.
2. If creating a console and user does not have PMGR privileges then error - User not privileged for this action.
3. Force us into the user context (0B0)
4. Resolve the pathname (WRSLV)
5. Get the namespace address, parent CCB and hash value (returned by WRSLV.
6. Make sure the user has write or append access to the directory.
7. If we are creating a network oriented file, make sure we are doing so in :NET else error - (3b0)

8. If file is too deep in directory depth, error = (200).

9. Calculate the number of DDEs that will be required by the new files FNB.

10. Search for the DDEs [JELLO in SOV10], if no space then error = No room in directory.

11. BLM filename from namespace onto the stack.

12. If link, (chain to [CLINK in NAME2], which will allocate a short FIB and a FLB; link them together; release all temporary memory used (i.e. FNB namespace (from GSMEM)) -- The system call is done.

13. a. If generic file or IPC, allocate a short FIB (if error = release FNB and die).

    b. If CPD or volume entry, allocate long FIB (if error = release FNB and die).

    c. Otherwise (normal files ...) allocate normal FIB (if error ...)

14. Place initial information in FIB (i.e. Time of creation, element size, EOF, first disk address, status, current/max sizes (if CPD).

15. Flag FIB as a FIB.

16. Flush FIB to disk.

17. If directory, increment parent's inferior directory count.

CREATE

SOV13

18. Kill the parent CCB.

19. Set up the initial ACL [JSACL]

20. Free up all temporary memory. -- The system call is done

5.4.2 ?RDB / ?WRB / ?SPAGE

?RDB \
?WRB) - IOCAL
?SPAGE /

Overlays:IOOV
IOOV1
Enter with AC2 = Address (TCB)

1. Set up system call flag by examining ?TSYS and setting offset SYSWD to 180 if ?SPAGE, 1 if ?RDB, or 0 if ?WRB.

2. Clear the physical I/O flag (offset PIOFL).

3. Map the parameter packet into the 70K, 72K slots (DM70)

4. Call IOVLD (which will validate and setup request (see below)).

5. Dispatch to appropriate enque routine:

NQCCB	-	disk
MQCCB	-	MTA
MCACB	-	MCA
LPBIO	-	LPB/LPD

6. On return, 0 --> 36o (CRSEG) and jump to PCALL.

7. (FSHR in module: IOOV1) If FCB is locked, restart the call processing later as SRDB call by jumping to SNTIO in SCPRC.

8. If user doing ?SPAGE has write access to the file, assume that the user will modify and set the modified bit in the FCB.

9. If shared block is not in core, convert system call into a SRDB call and jump into SCPRC at SNTIO.

10. If the user does not have write access to the file, write protect the shared page.

11. Update the map.

12. Update the CCB.

13. If more pages are to be read in, goto 9. Else unlock the CCB, unpend the TCB, zero CRSEG (3oo) and jump to PCALL.

5.4.3 ?PRDB / ?PWRB

Overlay: IOOV1

1. Set flag indicating physical I/O (-1 -> PIOFL).
2. Set read/write flag according to system call word.
3. Map Physical I/O (PIO) packet into 70,000 slot.
4. Validate and setup request (call IOVLD [see below]).
5. Decrement PSQCT. This is necessary because IOVLD, thinking that the system call is direct, has ISZed it one to many times. PIO calls are not direct calls and therefore we must compensate.
6. Set up Buffer Headers with data address, map word, # of blocks, starting address, TCB address and the read/write flag.
7. Enqueue the request (NQBHR) and wait for completion (BWAIT).
8. Remap in user packet so that error and status words can be passed back to the user.
9. Unlock the CCB (it was locked by IOVLD).
10. Return (the system call is complete).

5.4.4 IOVLD (module: IOOV) - Request validation and setup

This routine (called as a subroutine) will perform validation and setup for I/O requests.

1. Save the # of blocks, block # (high and low), last byte count, and status on the stack
2. Convert the user word address into a logical page # and save on stack.
3. If ?SPAGE make sure device block #, requested # of blocks and user word address is page aligned and that the shared I/O request is within the shared area.
4. If ?RDB or ?WRB to other host restart system call, with a CB, as either 177 or 176 respectively. (Network read/write reflection)
5. Validate channel number (0 < chn # < 63).
6. Calculate the CCB address and map it in. If the CCB does not exist, then error - File not open.

7. If this is a remote request, pass the call off to RMA

8. Try and lock the CCB. If we cannot then error - Channel in use.

9. If null disk I/O request error - Invalid parameter to system call.

10. Map in the FCB and verify that it matches the CCB. (Offset FBUID must match CBUID)

11. Initialize CCB parameters.

12. Modify the following bits:

If ?SPAGE	Set BPFSC (shared area changed) in PTBL Set BCBSH (shared request) in the CCB Set BCBSA (shared file) in the CCB
If ?WRB	Reset BCBSH (shared request) in CCB Set BCBC1 (write I/O) in CCB Set BFBMD (file modified) in FCB
If ?RDB	Reset BCBSH (shared request) in CCB. Reset BCBC1 (write I/O) in the CCB.

13. If ?SPAGE goto step 7 in ?SPAGE description (section 5.4.2)

14. Check the appropriate ACL

15. Charge for the I/O request

16. If request is to LPB/LPD or past LOT on MTA/MTB set BCBEO in CCB.

17. If target is unit CCB then set up request (MTA/MTB/MTC/MCA/LPB/LPD)
    (If ?RDB from LPB/LPD error - file read error)

18. Calculate # of pages referenced and validate each page.

19. If this is a modbits system, and we are doing a ?RDB, it is possible that the page we are reading into is shared. In this case, the page will be modified by the datachannel and the hardware modbits will not be aware of the change. So if this is a ?RDB, set the hardware page modified bit in the CME unconditionally (no harm if not shared)

20. Increment active call count to keep process from being swapped. Set 'Don't make virtual' bit.

21. Calculate I/O request priority. (377 if process is swappable else the I/O request priority is equal to the callers PNUF)

22. Return

5.4.5 ?OPEN

1. Make sure that the channel is not already in use. (This information is obtained by looking at the CCB offsets in the PEXTN.)

2. Resolve pathname through the directory structure.

3. If the file is not open any where else, create a FCB and copy the relevant information from the FIB. (If the file is not open, the FIB for the file will contain 0, else the file will contain a pointer to the FCB.)

4. Allocate a CCB and point it back to the FCB.

5.5 Key Diskworld page zero locations

The following locations in page zero (defined in SZERO.LS and STABLE.LS) are useful in examining the disk world. Some locations are pointers to chains, others are metering locations, counter or flag words.

SZERO.LS locations:

LCBP:	0	Pointer to current LCB, used by DSKIO
.MLCB:	-1	Pointer to a linked list of defined LCBs for the system
.ELCB:	0	Pointer to the end of the LCB list.
WFLAG:	0	180 - Base level waiting for system buffer

STABLE.LS locations:

NQBUF:	0	Word indicating if anyone waiting on an NQ block
FNQB:	-1	Pointer to head of free NQ block list
IORUN:	-1	IOCB running list head
IOFRE:	-1	IOCB free list head
CCBWQ:	-1	CCB wait queue head
RLCFL:	0	RUNLCB flag - non zero if there is something for RUNLCB to do.
BFLRU:	-1	List of free buffers, in least recently used order
BFMIN:	SCBFM	Minimum # of buffers
BWFLG:	0	Buffer wait flag - set by GSHEM when waiting for core manager to free a buffer cleared by core manager (@FREBF) when he successfully shrinks the system buffer pool
CACNT:	0	Count of allocated CACHE descriptors
CAHDP:	0	Pointer to head of descriptor LRU chain
CATLP:	CAHDP	Pointer to tail of descriptor LRU chain
CA64:	0	Map word for 1st page of descriptors
CA66:	0	Map word for 2nd page of descriptors

FCBCH: -1 FCB chain, chain of 8 word descriptors defining the physical memory blocks being used for FCBs

Metering locations:

BLKCH: 0 # of BLKIN calls (hi)
BLKCL: 0 " (lo)
BLKFH: 0 # of times BLKIN finds block on FCB (hi)
BLKFL: 0 " (lo)
BLKIH: 0 # of times BLKIN finds block with IOP (hi)
BLKIL: 0 " (lo)
NQBCH: 0 # of NQBHR calls (hi)
NQBCL: 0 " (lo)
NQBBH: 0 # of NQBHR calls with > 1 on Q (hi)
NQBBL: 0 " (lo)
NQCCH: 0 # of NQCCB calls (hi)
NQCCL: 0 " (lo)
NQCVH: 0 # of NQCCB calls with virtual CCB (hi)
NQCVL: 0 " (lo)
NQCIH: 0 # of NQCCB calls with IOCB available (hi)
NQCIL: 0 " (lo)
CARQH: 0 Count of CACHE requests (hi)
CARQL: 0 (lo)
CAHTH: 0 Count of CACHE hits (hi)
CAHTL: 0 (lo)
DWBKH: 0 # of disk blocks written (hi)
DWBKL: 0 " (lo)
DRBKH: 0 # of disk blocks read (hi)
DRBKL: 0 " (lo)
SWPSH: 0 # of swap ins (hi)
SWPSL: 0 " (lo)
SWPAB: 0 # of swaps aborted

BUFCN: 0 # of buffers on BFLRU
BUFTC: 0 Total # of buffers
NQCNT: 0 Count of active NQ blocks
IOTC: 0 IOCB total count
IOAC: 0 IOCB active count
IOMX: 0 Max value of IOTC
CCWC: 0 Count of CCBs waiting for IOCB
CCMX: 0 Max value of CCWC
NQBHC: 0 # of buffer headers NQed to UDBs

CHAPTER 6 - SYSTEM CALL PROCESSING
(updated for AOS rev 3.11)

6.1 Introduction

System calls under AOS are processed either by the ghost, the system, or a combination of the two. A system call such as ?SYSID is processed exclusively by the system, a call like ?PROC is first preprocessed by the ghost, then the ghost makes a second system call to get the system involved in the processing.

This chapter will concern itself almost exclusively with system call processing in the system. There is a separate chapter dealing with GHOST processing.

6.2 General flow of a system call

Entry into AOS is achieved by two methods. The first is by a direct SYC instruction performed by the GHOST. The second, the one discussed here, is through a module called SCALL which is bound into the user program by BIND. It is SCALLs function to decide whether the call is a GHOST call (150 of the system call word is 1) or a straight system call (0B0). If the call is a ghost call, SCALL passes a secondary system call to AOS that forces the process to "ENTER GHOST". At this point the GHOST (and the appropriate chapter in this manual) take over. If the call is a system call, AOS takes over. This is flowcharted on the next pages.

In this chapter, there is a list of the system calls (and a technique for finding which overlay processes a given system call) and a list of specific characteristics that some system calls possess.

6.3 Kernel system call processing

Entry to AOS from a user is always via an "SYC" type instruction which disables mapping, optionally pushes a return block, and then executes a JMP @2 instruction. There are three modes of entry:

1. Implicit -- Used for breakpoint processing

```
SYC 0,1 --> Debug breakpoint        (?SCLB)
SYC 0,2 --> Ghost breakpoint        (?SCLG)
```

2. Exclicit -- Extremely fast processing time (<50 microseconds) with fast entry into AOS. Note: SCL = SYC 1,1

SYC 1,1 code in AC0 identifies action required.

0	-- Enter Ghost	(?SCEG)
1	-- Remap primary page to ghost	(?SCRM)
2	-- not used	(?SCXS)
3	-- Schedule a task	(?SCST)
4	-- Exit Ghost	(?SCGX)
5	-- not used	(?SCSM)
6	-- enable LEF mode	(?SCLE)
7	-- disable LEF mode	(?SCLD)
10	-- return status of LEF mode	(?SCLS)
11	-- Address checker	(?SCAC)
12	-- reschedule Ghost request	(?SCGS)
13	-- not used	(?SCET)

Codes 2 and 13 are no longer used. They are left over from pre-rev 1.00 days when the user task scheduler was in the users address space. These codes were used in accessing and controlling the task scheduler.

3. TCB Request - The standard system call. Caller's TCB contains all relevant information. The call is made with a SVC (SYC 0,0).

In addition, system calls can be classified in the following manner:

1. Channel calls -- will eventually reference a user channel

?RDB ?WRB ?SPAGE ?GOPEN ?GCLOSE ?SOPEN ?GNFN ?SCLOSE ?CGNAME ?UPDAT ?ROPEN ?RCLOSE

2. Direct calls -- Processing of the system call will never pend, although the task making the call can pend. (No stack or CB is needed)

?RDB ?WRB ?DELAY ?MBTG ?MBFG ?MBTU ?MBFU ?RPAGE ?SPAGE ?SIGNAL ?ISEND ?IREC

3. Serial calls -- Only one of these system calls can be active at one time (HOGTBL)

?CREATE ?PROC ?INIT ?SLIST ?GLIST ?SACL ?GACL ?SOPEN ?FSTAT ?GNAME

4. Parallel calls -- A multitask process can have no other system calls active while this call is active.

?PROC ?MEMI ?CHAIN ?RPAGE ?SPAGE ?SSHPT

5. Other calls

6.4 AOS Modules involved in system call processing

6.4.1 SCALL -- bound into the user program

GHOST -- runs as an alternate context. Used for preprocessing and file oriented calls.

6.4.2 SCPRC -- system call processor. All system calls are processed through this module. It either process the call, dispatches the call to STRAP, or enqueues the request to the PEXTN (process table extension).

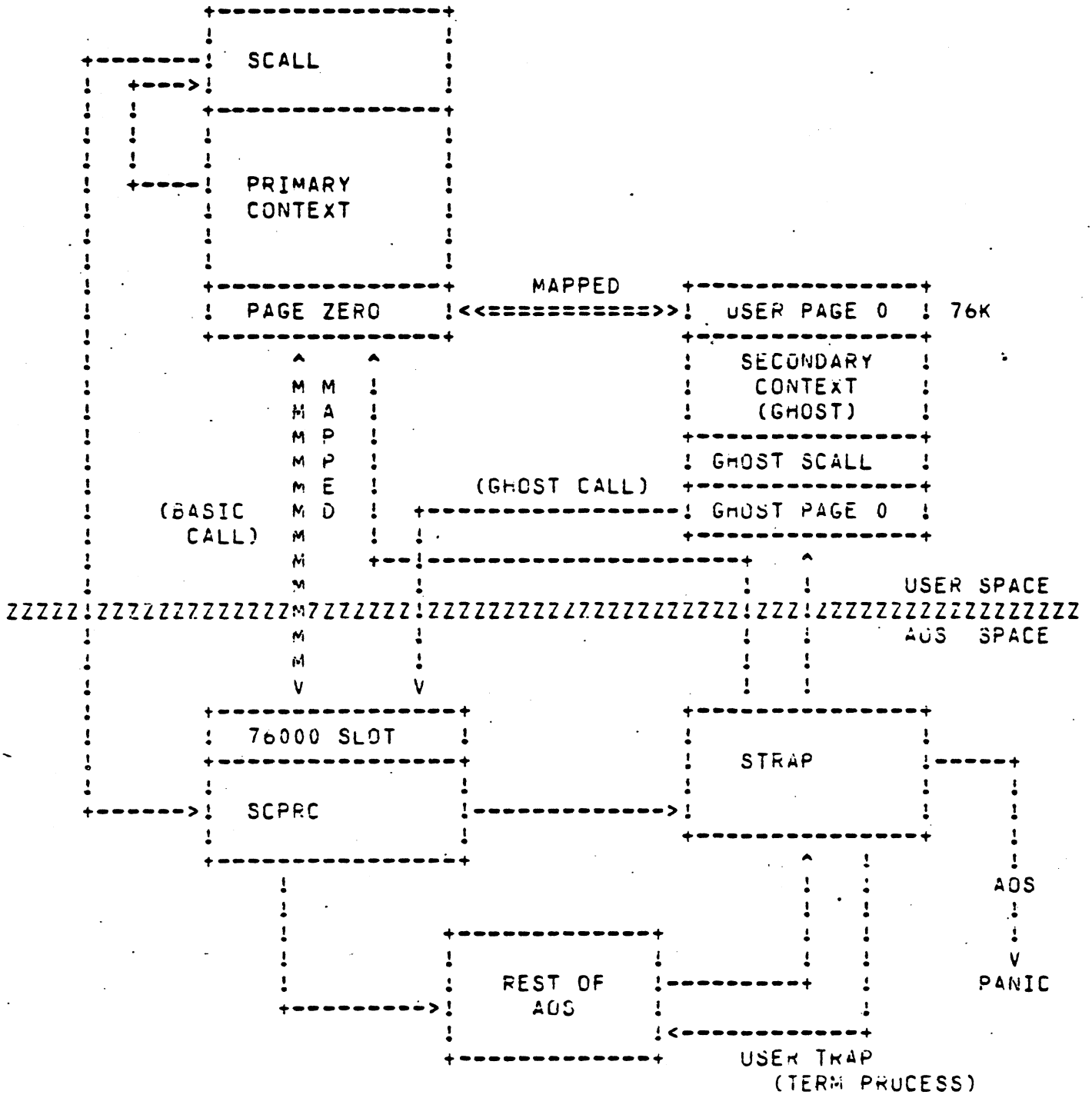
STRAP -- AOS trap handler. Implicit and Explicit calls are handled by this module.

6.4.3 SCHED -- The AOS scheduler. This picks up the enqueued requests from the PEXTNs, associates them with a CB and stack, and then jumps into the appropriate overlay for processing.

SCMOD -- Although most system calls are executed by overlays this overlay is special because it is involved with execution of direct system calls (i.e. those that can not pend)

The diagram provided on the following page is a graphic representation of the overall flow of a system call in AOS.

SYSTEM CALL PROCESSING INTERFACE

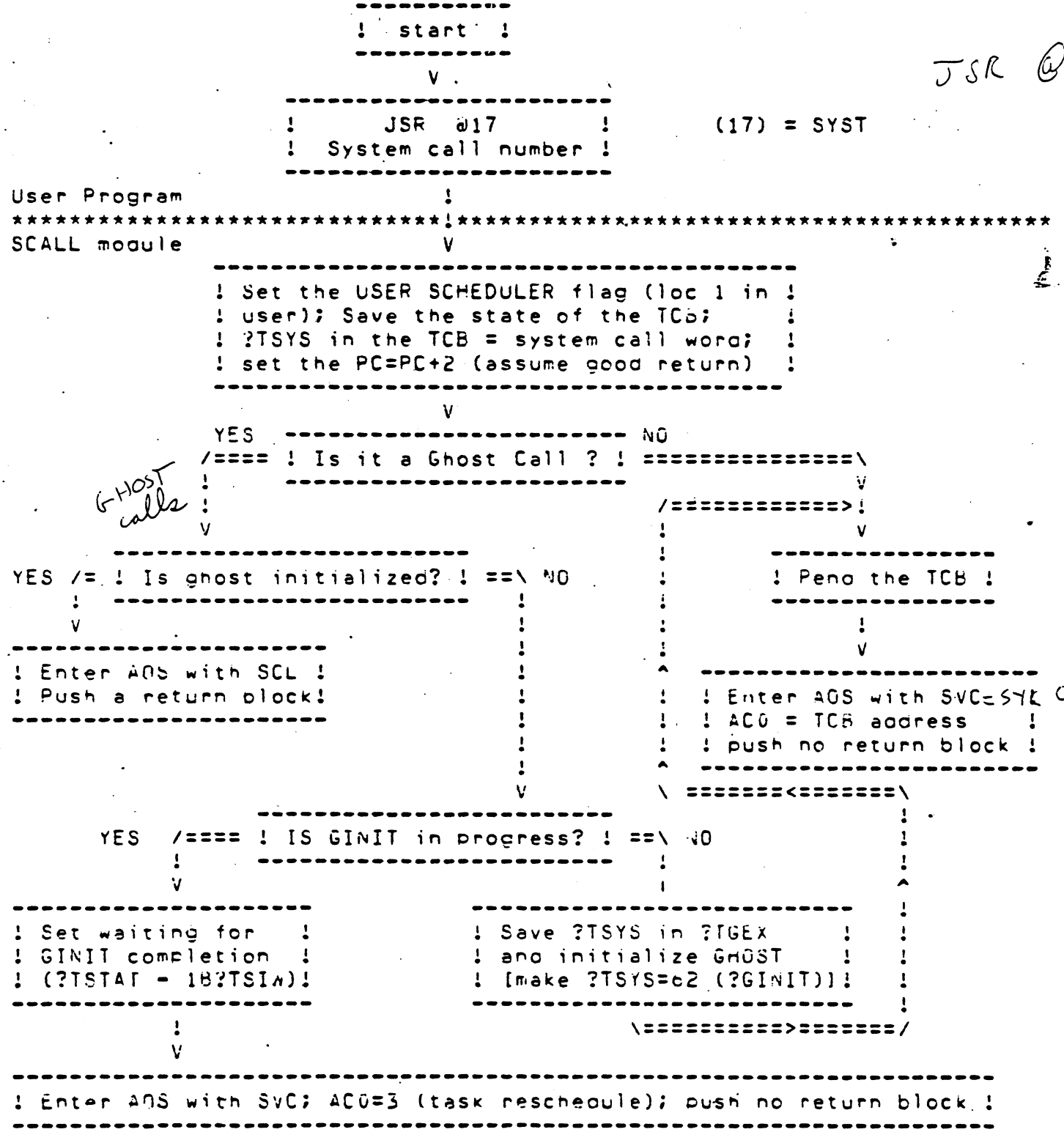


(IE. FOR BLOCK I/O DEVICES)

6.4.1 Module: SCALL

SCALL

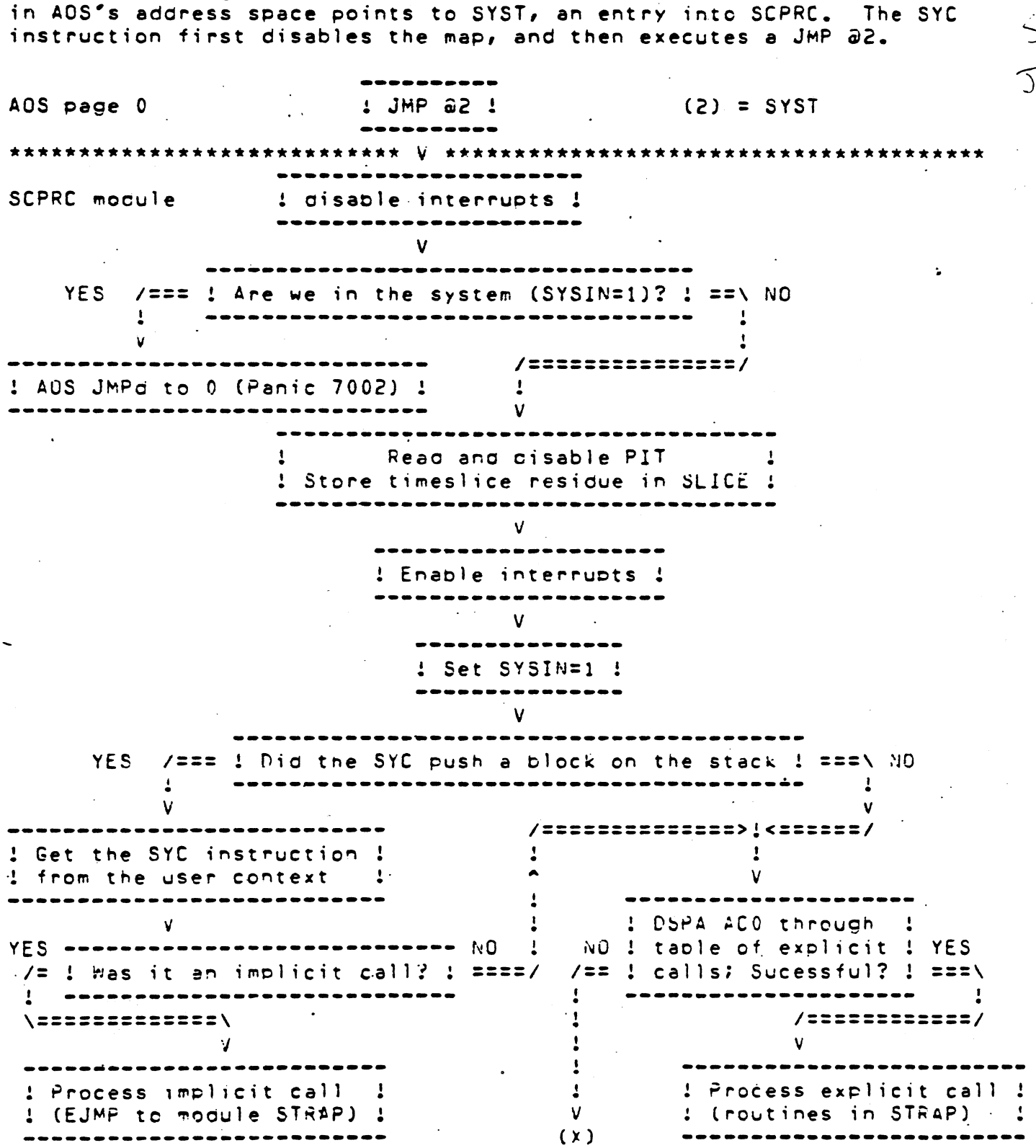
This module is automatically added into a user program at BIND time. The primary purpose of SCALL is to get the user into AOS system code. This module also defines the default STACK FAULT handler.

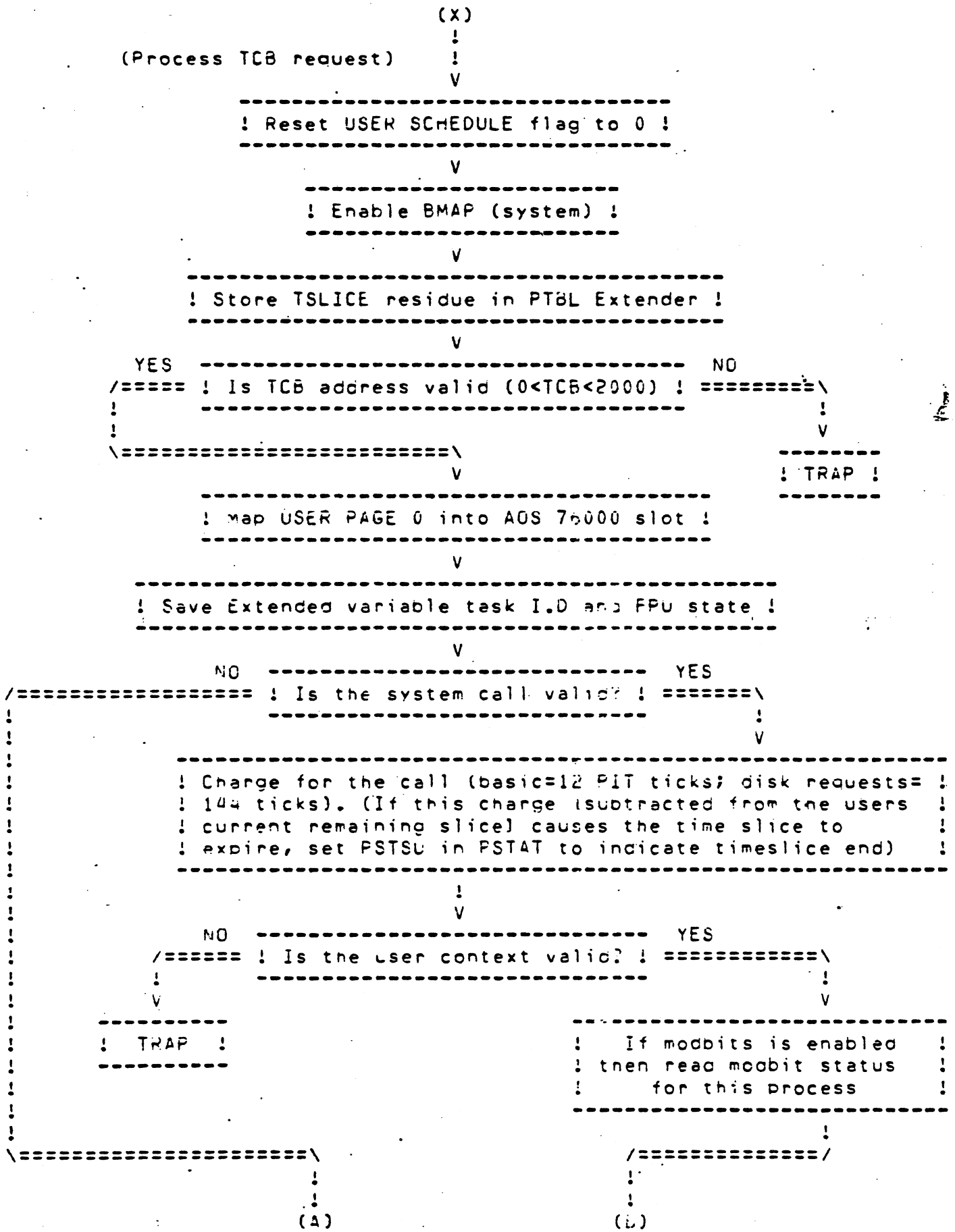


6.4.2 Module: SCPRC

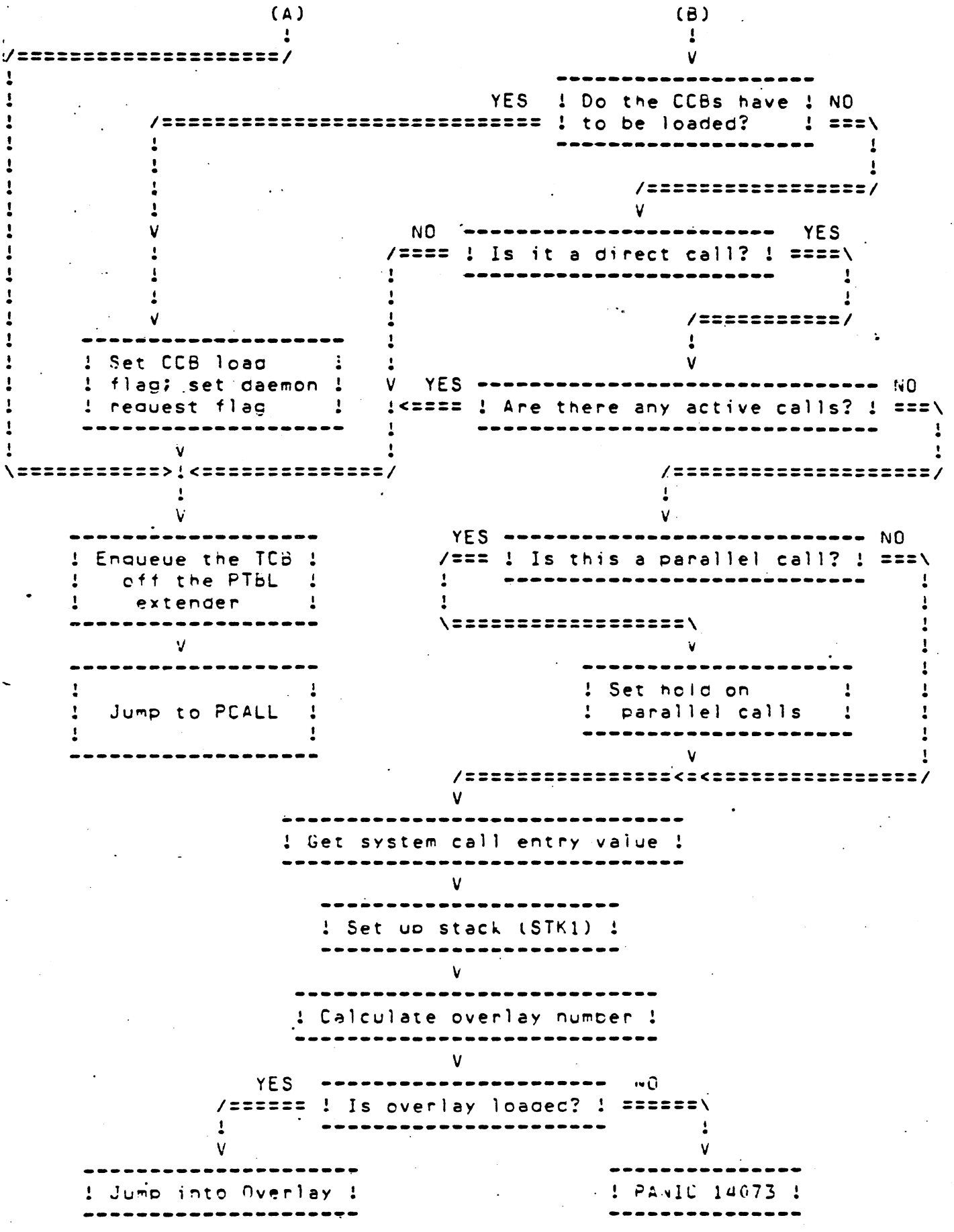
This module is part of the AOS kernel. All system calls are processed through it. At interrupt level 0 (no interrupts) location 2 in AOS's address space points to SYST, an entry into SCPRC. The SYC instruction first disables the map, and then executes a JMP @2.

SYC
JMP @2

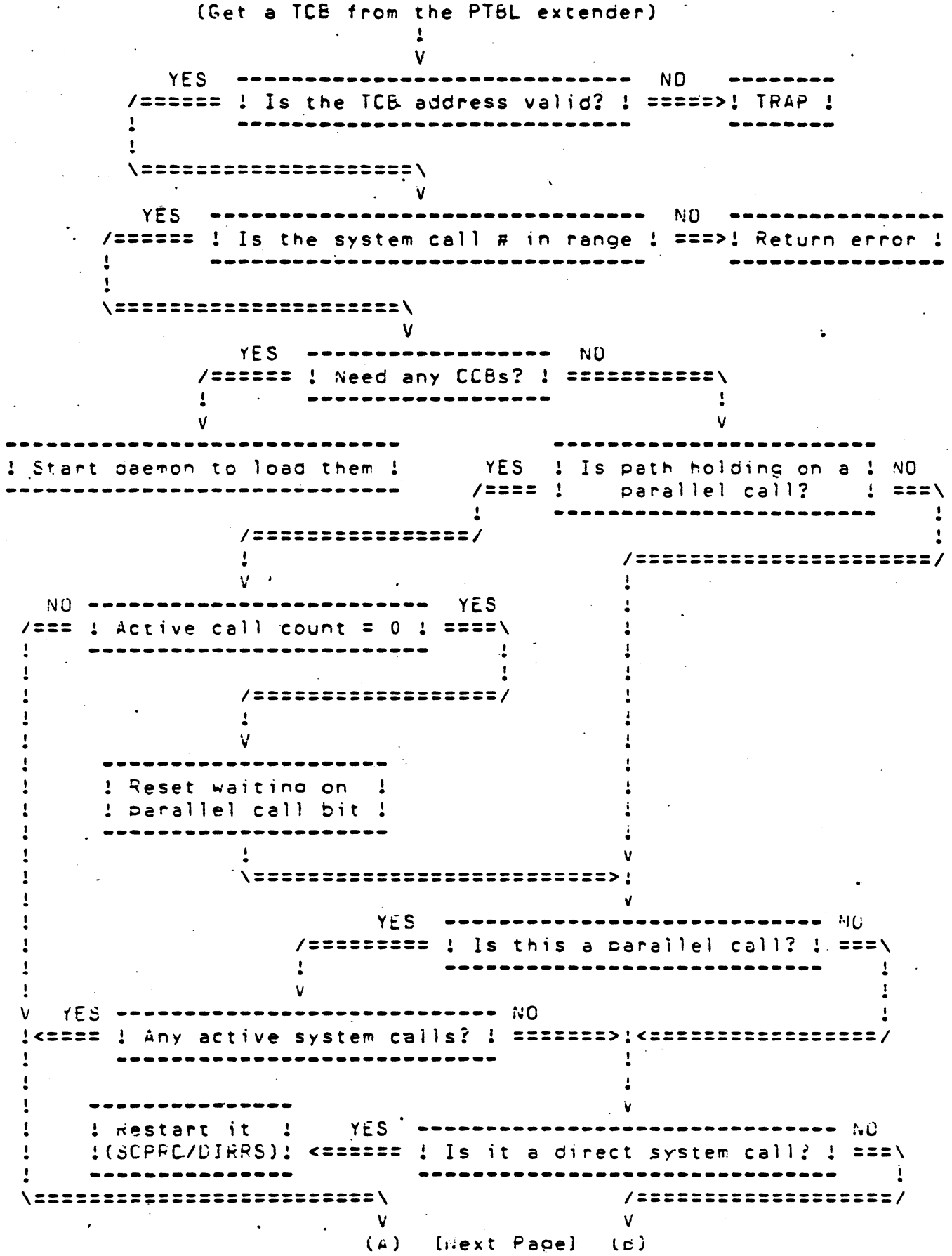


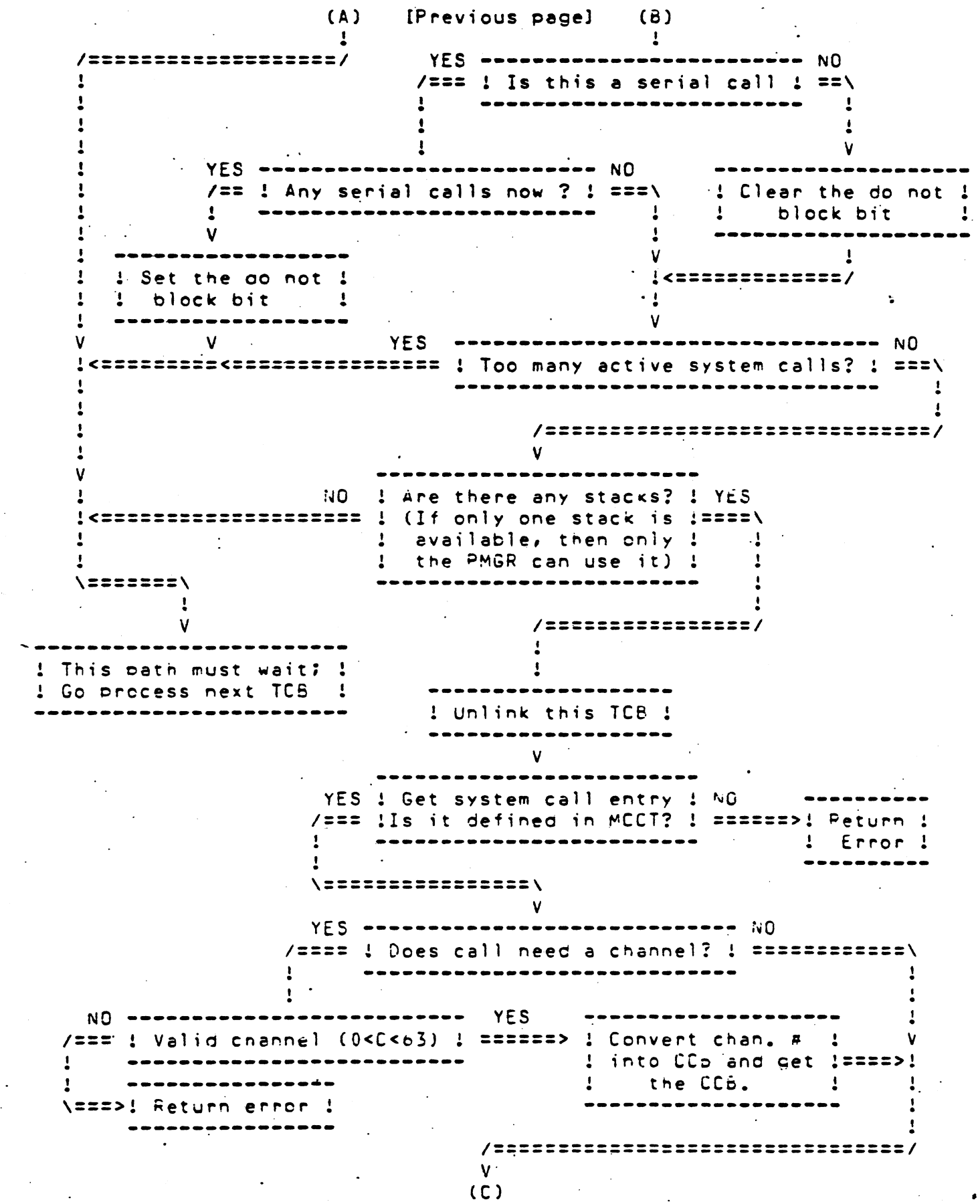


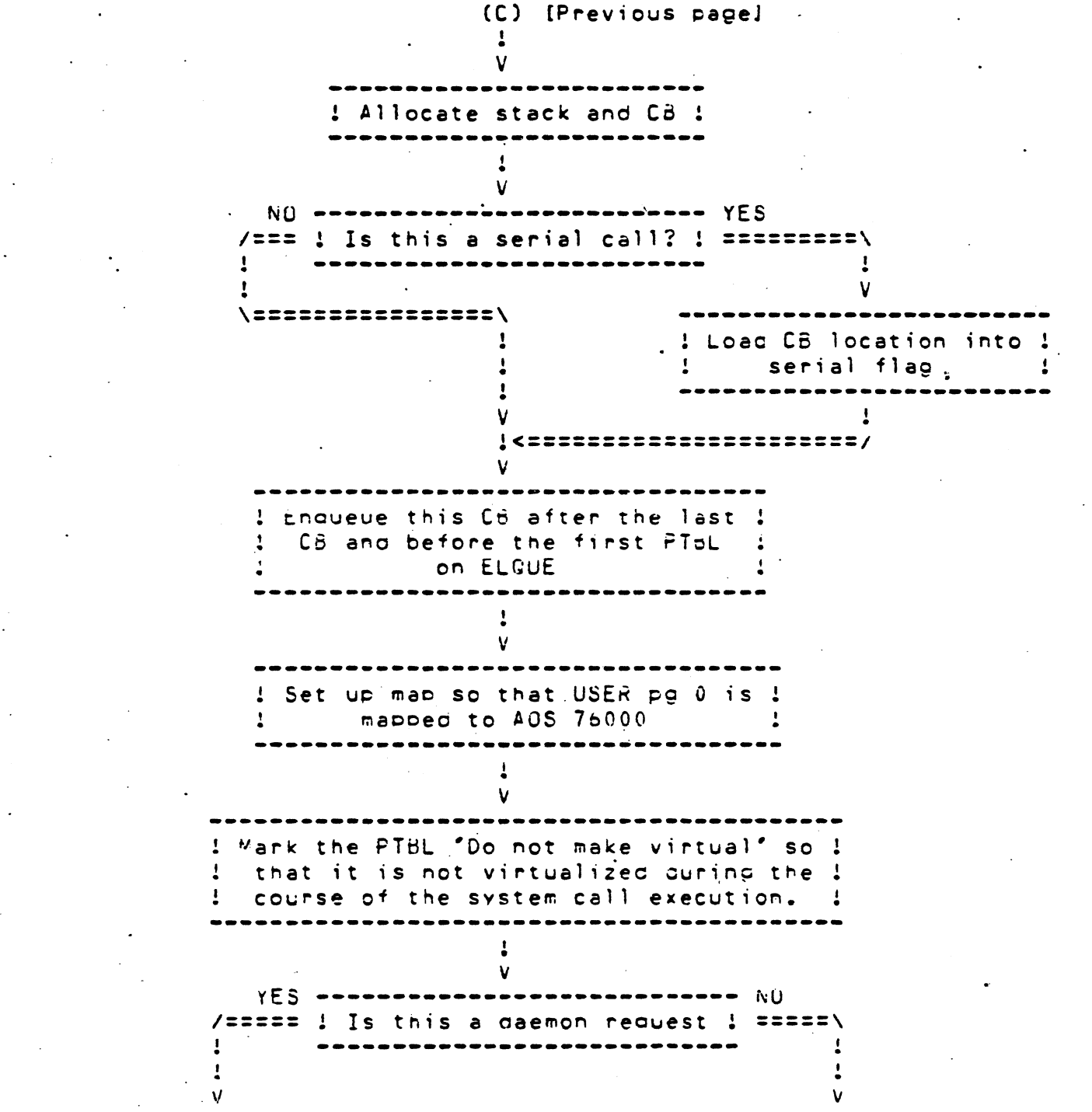
(cont. next page)


(A)
(B)
Do the CCBs have to be loaded?
YES
NO
Is it a direct call?
YES
NO
Set CCB load flag; set daemon request flag
Are there any active calls?
YES
NO
Enqueue the TCB off the PTBL extender
Is this a parallel call?
YES
NO
Jump to PCALL
Set hold on parallel calls
Get system call entry value
Set up stack (STK1)
Calculate overlay number
Is overlay loaded?
YES
NO
Jump into Overlay
PANIC 14073

6.4.3 Module: SCHED




(A)
[Previous page]
(B)
Is this a serial call
YES
NO
Any serial calls now ?
Set the do not block bit
Clear the do not block bit
Too many active system calls?
Are there any stacks? (If only one stack is available, then only the PMGR can use it)
This path must wait; Go process next TCB
Unlink this TCB
Get system call entry
Is it defined in MCCT?
Return Error
Does call need a channel?
Valid channel (0<C<63)
Return error
Convert chan. # into CCB and get the CCB.
(C)


(C) [Previous page]
Allocate stack and CB
Is this a serial call?
NO
YES
Load CB location into serial flag
Enqueue this CB after the last CB and before the first PTBL on ELGUE
Set up map so that USER pg 0 is mapped to AOS 76000
Mark the PTBL 'Do not make virtual' so that it is not virtualized during the course of the system call execution.
Is this a daemon request
YES
NO
Jump to individual daemon execution codes paths
Jump into the specific overlay required for the call

6.5 Tables

The following tables are found in SCPRC and are useful offsets for following system call paths

6.5.1 Vector table for SCLS and STR calls (offsets in STRAP)

```
        0                       ;LOW CODE
        ?SCET                   ;HIGH CODE
TRPVC:  EGST                    ;ENTER GHOST
        RMGST                   ;REMAP FOR GHOST
        -1
        UEXIT                   ;SCHEDULE A TASK
        GEXIT                   ;EXIT GHOST
        USMSK                   ;SET MASK
        LEFE                    ;LEF ENABLE
        LEFD                    ;LEF DISABLE
        LEFS                    ;LEF STATUS
        ADRCK                   ;ADDR CHECKER
        GSCHD                   ;GHOST RESCHEDULE REQ
        -1
```

6.5.2 System call dispatch locations

(NOTE: The overlay number for the following entry points can be calculated using DEDIT or SYSDMP with the following formula (?# is system call #):

= (@(MCCT+?#)%AND%77400)/400

This effectively takes the dispatch entry, masks out the right byte, and swaps the left into the right.)

```
MCCT:   CREATE                  ;0-  CREATE
        DELETE                  ;1-  DELETE
        RENAME                  ;2-  RENAME
        MEM                     ;3-  AVAILABLE MEMORY
        GCHAIN                  ;4-  CHAIN
        PRSTAT                  ;5-  PROCESS STATUS
        DPMGR                   ;6-  DEFINE PERPH PROC
            ; PAIRED CALL
        IOCAL                   ;7-  PRIMARY RDB PROC ADDR
        -1                      ;10- (RESERVED FOR READ BLOCK)
            ; PAIRED CALL
        IOCAL                   ;11- WRITE BLOCK
        -1                      ;12- RESERVED FOR .WRB
        PROC                    ;13- CREATE A PROC
        MEMI                    ;14- MEM INC
        TIMEQ                   ;15- QUEUE A TIME REQ
        INTAD                   ;16- DEFINE INT PROC ADDR
```

```
        MBTG                    ;17- MOVE BYTES TO GHOST
        MBFG                    ;20- MOVE BYTS FROM GHOST
            ; PAIRED CALL
        RMBTU                   ;21- PRIMARY CALL TO MOVE BYTES
        MBTU                    ;22- SECONDARY WITH STACK
            ; PAIRED CALL
        RMBFU                   ;23- PRIMARY CALL TO MOVE FROM
        MBFU                    ;24- SECONDARY WITH STACK
        DRSND                   ;25- IPC SEND
        IRECD                   ;26- IPC RECEIVE
        ILKUP                   ;27- IPC LOOKUP
        GRUNT                   ;30- GET RUNTIME STATS
        TABT                    ;31- ABORT CALL
        XLATE                   ;32- TRANS A TO A MAPPED ADDR
        CNGTY                   ;33- CHANGE PROC TYPE
        LINF                    ;34- USER WILL USE INFOS
        INFI                    ;35- INFOS PROCESS DECLARATION
        GTIME                   ;36- GET TOD
        STIME                   ;37- SET TOD
        SDAY                    ;40- SET DAY
        GDAY                    ;41- GET DAY
        IDEF                    ;42- DEV INTERRUPT DEFINE
        IRMV                    ;43- INTERRUPT REMOVE
        SSHPT                   ;44- SET SHARED PARTITION
        RPAGE                   ;45-  REL SHARED SLOT
        CDIS                    ;46-  DISABLE PROG CONT A
        CEBL                    ;47-  ENABLE CONT A
        DEBL                    ;50-  ENABLE MAPPED DEV
        DDIS                    ;51-  DISABLE MAPPED DEV
        STMAP                   ;52- SET MAP FOR USER DCH
        SUPROC                  ;53- CHANGE SUPERPROCESS MODE
        TABT                    ;54- TASK ABORT
        TRMPR                   ;55- TERMINATE A PROC
        GOPEN                   ;56- OPEN
        GCLOSE                  ;57- CLOSE A USER CHANNEL
            ; PAIRED CALL
        IOCAL                   ;60- SPAGE- SHARED READ
        SRDB                    ;61- SHARED READ PATH ON STACK
        IGHOS                   ;62- INIT GHOST
        SOPEN                   ;63- SHARED OPEN
        GPORT                   ;64- GET A PORT OWNER
        TPORT                   ;65- TRANSLATE A PORT NUMBER
        BLKPR                   ;66- BLOCK A PROCESS
        UBLPR                   ;67- UNBLOCK PROC
        PRIPR                   ;70- CHANGE PROC PRI
        SIGNAL                  ;71- SIGNAL THE SYSTEM
        GUNM                    ;72- GET A PROCESS'S USER NAME
        GSHPT                   ;73- GET SHARED PARTITION VALUES
        GHRZ                    ;74- GET CLOCK FREQ
        DIR                     ;75- DIR
        DINIT                   ;76- INIT AN LDU OR MTV
        FSTAT                   ;77- GET FILE STATUS
        DRLSE                   ;100- RELEASE AN LDU OR MTV
```

```
SLIST                   ;101- SET SEARCH LIST
GLIST                   ;102- GET SEARCH LIST
LOGGR                   ;103- MANIPULATE SYSTEM LOG
GBIAS                   ;104- GET BIAS FACTOR
SBIAS                   ;105- SET BIAS
IHIST                   ;106- INIT HISTOGRAM
KHIST                   ;107- KILL HISTOGRAM
COPEN                   ;110- GHOST SHARED OPEN
GNAME                   ;111- GET FULL PATHNAME
GNCP                    ;112- GET A CONSOLE PORT NUMBER
SUSER                   ;113- CHANGE SUPERUSER STATUS
SACL                    ;114- SET A FILE'S ACL
GACL                    ;115- GET A FILE'S ACL
PNAME                   ;116- PROCESS NAME <-> PID
-1                      ;117- RESERVED
FLUSH                   ;120- FLUSH SHARED PAGE TO DISK
-1                      ;121- GET A FILE'S ALIAS
GTACP                   ;122- GET ACP'S FOR A FILE
-1                      ;123- DELETE FILE AND ALL NAMES
GLINK                   ;124- GET LINK CONTENTS
GPRNM                   ;125- GET PROGRAM NAME
LOGEV                   ;126- LOG EVENT IN SYSTEM LOG
DADID                   ;127- GET FATHER'S PID
CPMAX                   ;130- SET CP DIR MAX SIZE
GNFN                    ;131- GET DIR'S NEXT FILE NAME
SPY                     ;132- MAP PERFORMANCE DATA
RDUDA                   ;133- READ USER DATA AREA
WRUDA                   ;134- WRITE USER DATA AREA
CRUDA                   ;135- CREATE USER DATA AREA
-1                      ;136- ASSOCIATE A FILE
-1                      ;137- DISASSOCIATE A FILE
-1                      ;140- GET NEXT FILENAME
SATP                    ;141- SET FILE ATTRIBUTES
IS.R                    ;142- IPC SEND/RECEIVE
BRKFL                   ;143- BREAK FILE
SYEBL                   ;144- ENABLE A SYNC LINE
SYDBL                   ;145- DISABLE A SYNC LINE
SYSND                   ;146- SEND DATA
SYRCV                   ;147- RECEIVE DATA
SYPOL                   ;150- DEFINE A POLLING LIST
SYEPL                   ;151- EN. A TERM FOR POLL.
SYDPL                   ;152- DIS. A  TERM FROM POLLING
SYGES                   ;153- GET SYNC LINE ERROR STATS

.IFE    EC
-1                      ;154- DEFINE EXTENDED CONTEXT
-1                      ;155- INIT. EXTENDED CONTEXT
-1                      ;156- RELEASE EXTENDED CONTEXT
.ENDC

SINFO                   ;157- GET SYSTEM INFORMATION
AMAP                    ;160- LOGICAL TO PHYSICAL MAP
SCLOSE                  ;161- CLOSE A SHARED FILE
RPAGS                   ;162- RPAGE WITH FLUSH
```

```
        CGNAM                   ;163- GET PATH FROM CHANNEL #
        SSID                    ;164- SET SYSTEM ID
        GSID                    ;165- GET SYSTEM ID
        DACL                    ;166- USER DEFAULT ACL
        CONX                    ;167- CONNECT
        DCONX                   ;170- DISCONNECT
        SERVE                   ;171- BECOME A SERVER
        RESIGN                  ;172- STOP BEING A SERVER
        MBTC                    ;173- MOVE BYTES TO CUSTOMER
        MBFC                    ;174- MOVE BYTES FROM CUSTOMER
        PCONX                   ;175- PASS A CONNECTION
        WRB2                    ;176- NETWORK WRB DEFLECTION
        RDB2                    ;177- NETWORK RDB DEFLECTION
        EWKUP                   ;200- EXTERNAL .IWKUP FUNCTION
        GREMA                   ;201- ACCESS THE NETWORK
        RNAME                   ;202- HOST ID FROM PATHNAME
        RSEND                   ;203- SERVER ?ISEND
        ITIME                   ;204- GET TIME IN INTERNAL FMT
        FNAME                   ;205- SERVER PATHNAME RESOLUTION
        HNAME                   ;206- HOST ID<->HOSTNAME
        RPORT                   ;207- NETWORK ?TPORT

        .IFE    DP
        -1                      ;210- DEMAND PAGING ONLY
        -1                      ;211- DEMAND PAGING ONLY
        .ENDC
        KIOFF                   ;212- DISAB KEYBOARD INTERRUPTS
        KION                    ;213- ENABLE KEYBOARD INTERRUPTS
        KWAIT                   ;214- WAIT FOR A KEYBD INTERRUPT
        KINTR                   ;215- SERV KEYDB INTERRUPT FNCT
        VCUST                   ;216- VERIFY CUSTOMER RELATION
        CTERM                   ;217- TERM CUSTOMER PROCESS
        CRYPT                   ;220- ENCRYPT/DECRYPT
        SHEBL                   ;221- ENABLE SYNC LINE FOR HDLC
        SHDBL                   ;222- DISAE SYNC LN USING HDLC
        SHSND                   ;223- SEND USING HDLC PROTOCOL
        SHRCV                   ;224- REC USING HDLC PROTOCOL
        SNAME                   ;225- LOCAL. OF PROC/QUEUE NAME
        PIODS                   ;226- PHYSICAL ?RDB
        PIODS                   ;227- PHYSICAL ?WRB
        IREC                    ;230- NON-DIRECT IREC
        ISEND                   ;231- NON-DIRECT ISEND
        UPDAT                   ;232- FLUSH FILE DESCRIPTOR
        RDPEN                   ;233- RESERVED OPEN (GHOST ONLY)
        RCLOSE                  ;234- RESERVED CLOSE (GHOST ONLY)
        APOK                    ;235- AP WCS IS LOADED CALL
        APINI                   ;236- AP INIT
        APMAP                   ;237- AP MAP
        APREL                   ;240- AP RELEASE
```

6.5.3 System Call attribute tables

Specific Special System call information. These bit tables have a bit for each system call. Now reserving space for 256. system calls.

```
                ; THESE BIT TABLES HAS A BIT FOR EACH SYSTM CALL
                ;  NOW RESERVING SPACE FOR 256. SYSTEM CALLS

                ; CHANNEL CALLS ARE:
                ;       ?RDB
                ;       ?WRB
                ;       ?SPAGE
                ;       ?SPAGE+1
                ;       ?GOPEN
                ;       ?GCLOSE
                ;       ?SOPEN
                ;       ?GNFN
                ;       ?SCLOSE
                ;       SYNC CALLS- ?SSND ?SRCV ?SPOL ?SDBL ?SGES
                ;                   ?HDBL ?HSND ?HRCV
                ;       ?CGNAM
                ;       ?COPEN
                ;       ?PRDB ?PWRB ?UPDAT
                ;       ?ROPEN, ?RCLOSE
                ;
                ; BIT=1 => CALL INVOLVES A CHANNEL
                ;
000500 CHNTB:   187+189
000000  0
000003  1814+1815
150000  180+181+183
000200  188
000100  189
003760  185+186+187+188+189+1810+1811          ;SYNC CALLS
050000  181+183
000000  0
035474  182+183+184+186+187+1810+1811+1812+1812 ;HDLC SYNC CALLS ?PXXX
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
```

```
                ; DEFINE CALL BIT TABLE FOR FAULT OF CHANNELS
                ;       ?RDB
                ;       ?WRB
                ;       ?SPAGE
                ;       ?SPAGE+1
                ;       ?GOPEN
                ;       ?GCLOSE
                ;       ?SOPEN
                ;       ?GNFN
                ;       ?SCLOSE
                ;       GCHAIN
                ;       ?ABTC
                ;       ?TABT
                ;       ?FSTAT
                ;       ?GNAME
                ;       ?GTERM
                ;       ?MEMI
                ;       SYNC LINES- ?SSND ?SRCV ?SPOL ?SEBL ?SDBL ?SGES
                ;                   ?HEBL ?HDBL ?HSND ?HRCV
                ;       ?COPEN
                ;       ?PRDB ?PRWB ?UPDAT
                ;       ?ROPEN ?RCLOSE

004510 CFLTB:   184+187+189+1B12
000100  189
000017  1B12+1B13+1B14+1B15
150001  180+181+183+1B15
000300  186+189
000100  189
007760  184+185+186+187+188+189+1B10+1B11      ;SYNC CALLS
050000  181+183
000000  0
075474  181+182+183+184+186+187+1B10+1B11+1B12+1B13
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
```

```
                ; DIRECT CALLS ARE:
                ;       ?RDB
                ;       ?WRB
                ;       ?DELAY
                ;       ?MBTG
                ;       ?MBFG
                ;       ?MBTU
                ;       ?MBFU
                ;       ?RPAGE
                ;       ?SPAGE
                ;       ?SIGNAL
                ;
                ; BIT=1 => CALL IS DIRECT
                ;
000505 DBTB:    1B7+1B9+1B13+1B15
153000  1B0+1B1+1B3+1B5+1B6      ;5-DRSND,6-IRECD
002000  1B5
100100  1B0+1B9
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0

                ; DEFINE BIT TBL FOR GLOBAL SYNCRONIZATION OF SYSTEM
CALLS                   ; REQUIRING LARGE AMOUNTS OF GSMEM.
                ;
                ; BIT=1 => LARGE GSMEM GETTER
                ;
100020 HOGTB:   1B0+1B11                ;CREATE, PROC
000000  0
000000  0
010003  1B3+1B14+1B15           ;SOPEN,INIT,FSTAT
060114  1B1+1B2+1B9+1B12+1B13   ;SLIST,GLIST,GNAME,SACL,GACL
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
```

```
                ; DEFINE BIT TBL FOR FILE SYSTEM CALL CHARGES
                ;
                ; BIT=1 => EXPENSIVE CALL
                ;
160020 SCRG:    1B0+1B1+1B2+1B11        ;CRE,DEL,REN,PROC
000400  1B7                     ;ILKUP
000003  1B14+1B15               ;GOPEN,GCLOSE
010007  1B3+1B13+1B14+1B15      ;SOPEN,DIR,DINIT,FSTAT
000314  1B8+1B9+1B12+1B13       ;COPEN,GNAME,SACL,GACL
000100  1B9                     ;GNFN
000000  0
010000  1B3                     ;CGNAM
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0


                ; DEFINE BIT TBL FOR PARALLEL CALLS
                ;
                ; BIT=1 => CALL IS PARALLEL
                ;
                ; E200 PARALLEL CALLS ARE:
                ;       ?PROC
                ;       ?MEMI
                ;       ?CHAIN
                ;       ?RPAGE
                ;       ?SSHPT
                ;       ?SPAGE
                ;       ?DCTX
                ;       ?ICTX
                ;       ?RCTX
                ;       ?ABTC
                ;       ?TABT
004030 CNTB:    1B4+1B11+1B12
000100  1B9                     ;?ABTC
006010  1B4+1B5+1B12            ;?SSHPT,?RPAGE,?TABT
140000  1B0+1B1
000000  0
000000  0
000016  1B12+1B13+1B14
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
000000  0
```

THIS PAGE ORIGINALLY WAS LEFT BLANK

CHAPTER 7 - INTERPROCESS COMMUNICATIONS AND CONNECTIONS (updated for AOS Rev. 3.11)

7.1 Introduction

In this chapter we will detail the mechanisms of both IPC and the Connection Manager. The user functionality of these facilities is well documented in the "AOS Programmer's Manual". Here we deal with the system aspects of these interactions, i.e. the code paths and databases which implement either interprocess communications or connections. We also describe how the system uses IPC to communicate with GHOST. PMGR and EXEC use of IPC is studied in chapters 9 and 10 respectively.

7.2 IPC

The system modules which effectively perform the sending/receiving of IPC messages are the two resident overlays IRECD and DRSND, plus the three disk based overlays IREC, ISEND and ISEN2. The ?IREC and ?ISEND calls are started as direct calls, branching to either IRECD or DRSND. If any situation would cause it to pend, the call is restarted as non-direct and will execute in IREC or ISEND/ISEN2. ISEN2 is chained to from ISEND in the event that spooling has to occur.

Additionally, some supporting IPC system calls are documented here : ?ILKUP, ?TPORT, and ?GPORT are coded in overlay IPC; ?GCPN is an entry point in overlay SSOV5; IPC entries are created by ?CREATE in module CREATE. All these overlays are disk based overlays.

7.2.1 IPC Initialization

IPC initialization for every new process (except CLI&T at system startup) is done by CIPCS in PROC2. The IPC spool file ":PROC:IPS.PID" is created if it does not already exist, and its CCB address is saved in the new process' process table. The spool file is then opened or re-opened and its CCB locked. The message count in block 0 is zeroed and the bit map in block 1 set up, and the CCB unlocked. Offset ?PIPC in the ?PROC packet has the address of an IPC message header or -1 if none. If any, the initial IPC message is sent via IIPC in ISEND. If caller is GHOST and indicated an IPC message in AC0, then it is sent via SIPC in ISEND (see GHOST to GHOST IPC, section 7.2.12).

spool files

7.2.2 The IPC Spool File

The IPC spool file is organized as follows :

BLOCK	WORDS	USE
0	0 - 255.	Directory
1	0 - 127.	Bit map
1	128. - 255.	Buffer
2	0 - 255.	Buffer
.		.
.		.
.		.
127.	0 - 255.	Buffer

7.2.2.1 Spool file directory

The 256. words of the spool file's block 0 are organized into 36. seven-word sets. The last 4 words are unused. The first word of the zeroth set (ISNE) contains the number of entries in the spool file. Initially, this is set to zero. The next six words of the zeroth set are unused. The next 35. sets contain space for up to 35. entries (ISMNE). That is, 35. messages may be spooled to a process before the error "File space exhausted" is issued due to directory overflow. These entries are time-ordered, top-down, and compressed. It is also possible to get the error "File space exhausted" if the message spooled is too large for the spool file. There is a maximum of 32,384. words (256.* 126.5) which can be allocated for messages in the spool file. Note that the spool file can become fragmented because the messages are not compressed.

A spool file directory block entry has the following format :

MESFL	= ?ISFL	system flags
MEUFL	= ?IUFL	user flags
MEOPH	= ?IOPH	origin port number (hi)
MEOPL	= ?IOPL	origin port number (low)
MEDPN	= ?IDPN	destination port number
MELTH	= ?ILTH	message length (in words)
MEPTR	= ?IPTR	word pointer to message text in buffer (user buffer before the call, spool file buffer after spooling)

When a message is spooled, ISNE is incremented and the header is written to the set pointed to by ISNE. The message, if any, is placed somewhere in the spool file buffer as determined by the bit map. Offset MEPTR in the directory entry is updated accordingly. When a ?IREC call examines the spool file, it looks for the first matching entry, starting at set one. If found, the entry is deleted, ISNE is decremented, all subsequent entries are moved up one block, and if there was a message, its space is freed in the bit map. That is, directory space is re-used by relocating all entries following the one just released. Buffer space is re-used in a different way : the algorithm allocates space to messages longer than one disk block starting at the first free full block in the spool file, downwards, using as many blocks as needed and stopping on a 16. words boundary; for messages shorter than one disk block, the bit map is scanned until enough contiguous free 16. words nuggets are found to hold the message.

To avoid looking at the spool file just to find out if it is empty, a bit in the spool file CCB, BCBMS, is set to indicate when the spool file is non-empty.

7.2.2.2 Spool file bit map

1 bit = 16. word nugget

1 word = 256. word = full disk block

Allocation is from left to right in a word. A set bit means the corresponding nugget is free. ISEN2 will zero the appropriate bits when it allocates space to hold a message in the spool file.

The bit map for the spool file is initialized as follows by PROC2 at process creation time :

BITMAP WORD	CONTENTS	BLOCK ALLOCATION
0	0	directory
1	377	1st half:bitmap, 2nd half:free
2	-1	free
.	-1	.
.	-1	.
.	-1	.
127.	-1	free

7.2.2.3 Spool file sample

The following is an output from 'X DISPLAY :PROC:IPS.003'. It is the spool file for the EXEC. Presently, there is one message in the file. There are remnants of many previous messages.

The spooled message has an entry of 0/0/0/0/200/20/620 . 620 is a pointer to the message in this file. 20 is the length of the message. This message is to the EXEC GHOST (port 200) from the system (port 0/0). The message itself, at location 620 in the spool file, is all zeroes. See "GHOST-to-GHOST IPC".

```
000000 000001 000000 000000 000000 000000 000000 000000 000000
000010 000000 000000 000000 000200 000020 000620 000000 013055
000020 000000 000403 000005 000000 000000 000000 013043 000000
000030 000403 000005 000000 000000 000000 013020 000000 000403
000040 000005 000000 000000 000000 013020 000000 000403 000005
000050 000000 000000 000000 013020 000000 000403 000005 000000
000060 000000 000000 013020 000000 000403 000005 000000 000000
000070 000000 013020 000000 000403 000005 000000 000000 000000
000100 013020 000000 000403 000005 000000 000000 000000 013020
000110 000000 000403 000005 000000 000000 000000 013020 000000
000120 000403 000005 000000 000000 000000 013020 000000 000403
000130 000005 000000 000000 000000 013020 000000 000403 000005
000140 000000 000000 000000 013020 000000 000403 000005 000000
000150 000000 000000 013020 000000 000403 000005 000000 000000
000160 000000 013020 000000 000403 000005 000000 000000 000000
*********
000400 000000 000277 177777 177777 177777 177777 177777 177777
000410 177777 177777 177777 177777 177777 177777 177777 177777
000420 177777 177777 177777 177777 177777 177777 177777 177777
000430 177777 177777 177777 177777 177777 177777 177777 177777
000440 177777 177777 177777 177777 177777 177777 177777 177777
000450 177777 177777 177777 177777 177777 177777 177777 177777
000460 177777 177777 177777 177777 177777 177777 177777 177777
000470 177777 177777 177777 177777 177777 177777 177777 177777
000500 177777 177777 177777 177777 177777 177777 177777 177777
000510 177777 177777 177777 177777 177777 177777 177777 177777
000520 177777 177777 177777 177777 177777 177777 177777 177777
000530 177777 177777 177777 177777 177777 177777 177777 177777
000540 177777 177777 177777 177777 177777 177777 177777 177777
000550 177777 177777 177777 177777 177777 177777 177777 177777
000560 177777 177777 177777 177777 177777 177777 177777 177777
000570 177777 177777 177777 177777 177777 177777 177777 177777
```

```
000600 000000 000404 000000 000405 052103 044137 046111 051524 TCH_LIST
000610 026100 046120 041000 000012 177777 177777 000006 000013 ,@LPB <12>
*********
000640 051524 040522 052054 046120 052054 040114 050102 000000 START,LPT,
                                                                @LPB
*********
000660 044105 040504 042522 051454 040114 050102 026062 000000 HEADERS,
                                                                @LPB,2
000670 047516 030000 000000 000000 000000 000000 000000 000000 ON0
000700 041517 047124 044516 052505 026100 046120 041000 000000 CONTINUE,
                                                                @LPB
*********
000720 042516 040502 046105 026100 041517 047067 000000 000000 ENABLE,
                                                                @CON7
*********
000740 042516 040502 046105 026100 041517 047070 000000 000000 ENABLE,
                                                                @CON8
*********
000760 042516 040502 046105 026100 041517 047071 000000 000000 ENABLE,
                                                                @CON9
*********
001000 051524 040522 052054 046120 052054 040114 050102 000000 START,LPT,
                                                                @LPB
*********
001020 051524 040522 052054 041101 052103 044137 047525 052120 START,
                                                                BATCH_OUTP
001030 052524 026100 046120 041000 000000 000000 000000 000000 UT,@LPB
001040 051524 040522 052054 041101 052103 044137 046111 051524 START,
                                                                BATCH_LIST
001050 026100 046120 041000 000000 000000 000000 000000 000000 ,@LPB
001060 044105 040504 042522 051454 040114 050102 026062 000000 HEADERS,
                                                                @LPB,2
*********
001100 041517 047124 044516 052505 026100 046120 041000 000000 CONTINUE,
                                                                @LPB
*********
001120 041517 047124 044516 052505 026061 000000 000000 000000 CONTINUE,1
*********
001140 053105 051102 047523 042400 000000 000000 000000 000000 VERBOSE
*********
001160 000000 000404 000000 000405 000000 000000 000000 000000
*********
```

7.2.3 Outstanding receive entries

When a process issues ?IREC and the corresponding ?ISEND cannot be found (but the origin PID does exist), then parts of the ?IREC header are linked onto the end of the "Virtual Outstanding Receive Request" (VORR) chain. This chain is in 'GVMEM' space, originating at offset PIORR of the receiver's process table. The address stored at PIORR is a virtual address in the following format :

bits 0-5 block index (word offset/16.)
bits 6-15 physical page #

The chain is terminated by a 0. When any database in this chain is accessed, the IPC lock is set (see IPC notes, section 7.2.13).

Format of an outstanding receive entry:

OLNK	link word (must be offset zero)
OOPH	origin port number high (HID/PID)
OOPL	origin port number low (16 bit port #)
ODPN	destination port number
OLTH	buffer length (in words)
OADDR	pointer to user buffer
OBUFH	pointer to user header 1B0 = 1 => GHOST made ?IREC call
OTCB	user TCB address
OSFL	system flag word

7.2.4 The ?ISEND modules

Overlay entry points in DRSND, ISEND and ISEN2 :

DRSND user entry.

ISEND entry point for non-direct call.

JSEND ISEND entry for ?IS.R and connection support. The privilege checks are skipped.

SIPC send a termination IPC message from the system.

IIPC send an initial IPC message; this is used by PROC2 to send a process its initial IPC message.

SENDR implements ?RSEND call : validates receiver's PID taken from left byte of DPL.

ISEN2 logic to spool an IPC message.

7.2.4.1 ISEND logic

Unless the caller is GHOST, the IPC privilege for the user is checked. The header is then moved to the stack as well as the sender's PID. The destination local port and the receiver's PID are checked for validity and the full destination and origin ports are set up and stored on the stack. The receiver's process table address is fetched and saved on the stack too. At this point the receiver's spool file CCB is locked; if it was already locked we pend waiting for its release. The message buffer in the receiver's address space is examined to make sure the entire area is defined in the user's unshared context.

The receiver's chain of VORR's is then scanned until an implicit or explicit match is found. If found, the matching VORR is unlinked from the chain, and the memory released via RVMEM. If no match is found, we chain to ISEN2 to spool the message.

If the receiver is eligible, its buffer is mapped into slots 70 & 72, the sender's buffer into slots 64 & 66 and the message is moved. The receiver's task is readied and the receiver's process unblocked if blocked.

If the receiver is swapped, the message is moved to the swapfile, the process unblocked if blocked, and the Core manager is woken up.

If the receiver is being swapped in or out, we pend waiting for the swap to be completed and try again.

In the case where the sender or the receiver is the system, the ISEND logic is basically the same. Validity checks are simplified and no mapping of message buffers in system space is required.

7.2.4.2 DRSND logic

The code in DRSND is just about the same as in ISEND as the functionality is similar. The big difference is that upon encountering any condition which would cause the path to pend, DRSND restarts the call as non-direct by changing the value of the system call word in the user's TCB.

Hence if the receiver's IPC lock is set, or no matching outstanding receive is found, or the receiver is ineligible, the call will be restarted for execution in ISEND.

When DRSND finishes processing, it returns to SMON1 instead of PCALL (ISEND returns to PCALL as do all CB system calls). Direct calls return to SMON1.

7.2.4.3 ISEN2 logic

When a message has to be spooled, the receiver's spool file directory is accessed, read in and checked for room for another entry. The length of the message is examined to determine how many full disk blocks and how many 16. word nuggets are needed to hold it. The bit map is fetched, and if the message requires one full disk block or more, we look for the first free full block and allocate as many full blocks and as many 16. word nuggets as we need. The IPC header is then stored as the next entry in the spool file directory. The full blocks part of the message is written out to the spool file via an internal write block routine (write block to system buffer); the last block is read in, the last part of the message mapped from the sender's address space to slot 64, the data moved and a request enqueued to write the block back to disk.

If the message requires less than one full disk block, each bit map word is considered sequentially until a sufficient set of contiguous free bits is found within one word. In each word, a search is made from left to right to find the first free bit, then if enough free bits follow the space is allocated, the block read in, the message mapped to slot 64, the data moved and a request enqueued to write the block back to disk.

7.2.5 The ?IREC modules

Overlay entry points :

IRECD user entry

IREC entry point for non-direct call

IS.R user entry point (non-direct call)

7.2.5.1 IREC logic

First the header is moved to the stack as well as the fetched receiver's process table address. The destination port is validated and the message buffer in the receiver's address space is examined to make sure the entire area is defined in the user's unshared context.

If there is a message in the receiver's spool file, the directory block is read in and its entries examined looking for a port match. If found, each of the entries after the matching one is moved up, the bit map is brought in and updated to free the area occupied by the message, and the message itself is read in and moved to the user space in the same way it was written out by ISEN2.

If no match is found but the sender's PID does exist, a VORR is built out of GVMEM space and linked onto the receiver's chain of VORR's hanging off its process table.

7.2.5.2 IRECD logic

The code in IRECD is also very similar to the code in IREC. Here again, the call is restarted as non-direct whenever a path would have to pend : if the spool file CCB lock is set, or the spool file is non empty, or we cannot get a GVMEM area for the VORR immediately.

7.2.5.3 IS.R logic

This system call performs an IPC send which, if successful, is followed by a receive with the same ports. The packet is a regular ?ISEND packet followed by two words specifying the length and location of the receive buffer. The call is non-direct and executes entirely in the ISEND and IREC overlays.

First a VORR is allocated, so that if the call is restarted, the IPC message does not get sent more than once. The packet is then moved to the stack, the caller's IPC privilege is checked and the send issued via entry JSEND, followed right away by the receive. Any error condition will go through a path which releases the memory acquired for the VORR.

7.2.6 The IPC module

Overlay entry points :

ILKUP : user entry for ?ILKUP

SLKUP : system entry for ?ILKUP, name in user space

VLKUP : " " " " " " system "

GPORT : user entry for ?GPORT

IGPOR : system entry for ?GPORT

TPORT : user entry for ?TPORT

RPORT : reserved for future use

RSEND : " " " "

MTSF : IPC move to swap file routine

7.2.6.1 Port look up logic

ILKUP takes a pathname, resolves it and checks for an IPC type entry with that name. If it finds one, it returns the port number associated with the name. If no entry is found, or the entry is not an IPC type entry, an error is returned. The system entry points share the same code paths, the difference being that the input AC's are passed on the stack instead of in the user's TCB.

7.2.6.2 Port translation logic

GPORT and TPORT take as input a global port number, and return the PID of the process who the port number is assigned to, plus his local port number. A PID of zero means the system owns the port. The system entry point gets its input from the AC's on the stack instead of in the user's TCB.

Since in Rev. 3 global format of the IPC ports is HID/PID in word 1 and local port # in word 2, this call is rather trivial and simply does a good return!

7.2.6.3 Move to swap file logic

If there is no message the header only will be moved. The message and header addresses are made byte pointers and the data is transferred via IMBTU after setting proper flags for Ghost. The receiver's process table is then retrieved, the process unblocked and readied and, when swapped in, the appropriate TCB will be unpended.

7.2.7 ?GCPN logic

The entry point for this call, GNCP, is defined in module SSOV5. This call takes a PID or a process name and returns the port number of that process' console, or an error if the process has no console. After the usual checks are performed, the process table address is retrieved and the process console port number in offsets PCONH and PCONL returned.

7.2.8 IPC entries (IPC as a communications device)

If two processes would like to communicate with each other, they may create and open an IPC entry and perform ?READ and ?WRITE to this entry in order to pass data to each other. These two ghost calls are actually prompting their ghosts to issue ?ISEND and ?IREC calls between the two processes.

GOV0, GOV1, and GOV2 are the ghost overlays which process the ghost calls calls ?OPEN, ?READ, and ?WRITE. These are the paths which implement IPC as a communications device. The highlights of this code which effect IPC will follow.

Suppose processes A and B want to communicate on port P. A and B must issue ?OPEN with the following parameters:

?ISTI = ?OFCE (if file does not exist, create and then open)
?ISTO = ?FIPC (IPC port entry)
?IFNP = byte pointer to "P"

Suppose A issues this create/open before B. When GHOST_A creates file P, it assigns a ghost port, "PORT_A". When GHOST_A opens file P, it issues a ?GOPEN which returns PORT_A as a parameter. GHOST_A then issues an ?IREC to receive from any port on PORT_A.

Now B opens file P. GHOST_B does a ?GOPEN retrieving PORT_A. GHOST_B selects a local port, PORT_B, and issues ?ISEND from PORT_B to PORT_A. After this is received by GHOST_A, both ghosts are informed of each other's port. Subsequent ?READ or ?WRITE calls will invoke ?IREC or ?ISEND between these two ports.

WARNING : If A issues ?CREATE and then ?OPEN to establish this IPC entry P, then P will be assigned a user port number which his ghost cannot send from. Thus, it is essential to issue the ?OPEN with the create option to assure that a ghost port is assigned to P.

7.2.9 Databases offsets definitions

Often, the same information is present in several IPC databases, referenced by different offset names. To assist one looking at the system modules, the following charts illustrate which offsets correspond.

For example, a user who issues ?ISEND, has a packet with an offset of ?IDPN. In DRSND/ISEND this is placed on the stack at offset 'DPN'. After port conversion, this word on the stack is referenced as 'DPN'. If this entry is spooled, this same word is stored at offset 'MEDPN' of the spool file entry.

A user who issues ?IREC, has a packet with an offset of ?ISFL. This is placed on the stack by IRECD/IREC at offset 'SFLAG', and if necessary, put in an outstanding receive entry at offset 'OSFL'. If this entry is accessed by DRSND/ISEND, it will be placed on that stack as 'RSFL'.

7.2.9.1 DRSND/ISEND definitions

Header	Stack	Converted	Spool file
?ISFL	SFLAG		MESFL
?IUFL	UFLAG		MEUFL
?IDPH	DPH		
		DPN	MEDPN
?IDPL	DPL		
		OPL	MEOPL
?IOPN	OPN		
		OPH	MEOPH
?ILTH	LTH		MELTH
?IPTR	MESA		MEPTR

7.2.9.2 IRECD/IREC definitions

Header/user	Stack	VORR entry	DRSND/ ISEND stack
?ISFL	SFLAG	OSFL	RSFL
?IUFL	UFLAG		
?IOPH	OPH		
?IOPL	OPL	OOPN	
?IDPN	DPN	ODPN	
?ILTH	LTH	OLTH	
?IPTR	BUFA	OADDR	UPTR
AC2	UAC2	OBUFH	UBUFH

7.2.10 The IPC ports

The IPC port numbers format is defined as follows :

Sender's packet:

DPH	D-host ID / D-PID
DPL	D-port
OPN	O-port

Receiver's packet:

OPH	O-host ID / O-PID
OPL	O-port
DPN	D-port

Port 0 is reserved. It is often used by ISEND/SIPC. Ports 1-177 are available for user ports. Ports 200-377 are GHOST ports. The PMGR uses ports 1-377 .

A user can receive on port 0 and match any user port. Ghost cannot receive on port 0. The PMGR can receive on port 0 and match any port.

The caller (sender or receiver) submits a packet when making his system call, including two ports. These ports are initially checked for validity by routine CHKPOR in resident module SGSub. Next, some of this information is converted. The ports are then in a common format : DPL-DPH-OPL-OPH. Any messages sent by a process to itself or received by itself (bit ?IFSTM or ?IFRFM selected) have converted port numbers of -1.

These converted port numbers are moved to the spool file or virtual outstanding receive chain as necessary. Comparisons are made between two sets of converted port numbers to see if ports match.

7.2.10.1 Valid ports

In the following tables, S = "Sender's PID"; R = "Receiver's PID". The "/" separates left byte from right byte.

"ISEND" means either DRSND or ISEND depending on whether the ?ISEND call is executing as direct call or not. The same stands for "IREC", which can be either IRECD or IREC.

To keep things simple, we assume HID = 0.

			Valid ISEND ports			__Converted ports__		
Entry	Caller	SFLAG	OPN	DPL	DPH	DPN	OPL	OPH
ISEND	user	0	1-177	0-377	0/R	0-377	1-177	0/S
ISEND	GHOST	0	200-377	0-377	0/R	0-377	200-377	0/S
ISEND	PMGR	0	1-377	0-377	0/R	0-377	1-377	0/S
ISEND	any	IFSTM	any	any	any	-1	-1	-1
SIPC	system	any	any	0-377	0/R	0-377	OPN	0
IIPC	system	any	any	any	any	-1	-1	-1
IS.R	same as ISEND ...							

			_Valid IREC ports__			__Converted ports__		
Entry	Caller	SFLAG	DPN	OPL	OPH	DPN	OPL	OPH
IREC	user	0	0-177	0-377	0/S	0-177	0-377	0/S
IREC	GHOST	0	200-377	0-377	0/S	200-377	0-377	0/S
IREC	PMGR	0	0-377	0-377	0/S	0-377	0-377	0/S
IREC	any	IFRFM	any	any	any	-1	-1	-1

7.2.10.2 Port matching rules

The following two conditions must be met in order to have a match of ports. Here, we are looking at the converted port numbers.

First condition :

A. IREC_DPN = ISEND_DPN

or

B. IREC_DPN = 0 and receiver = user and 0 <= ISEND_DPN <= 177

or

C. IREC_DPN = 0 and receiver = PMGR and 0 <= ISEND_DPN <= 377

Second condition :

A. IREC_OPL = ISEND_OPL

or

B. IREC_OPL = 0

7.2.11 Initial IPC and termination IPC

If ?PIPC <> -1 in the caller's ?PROC packet, then there is an initial message to be sent (via ISEND/IIPC) so that a "Receive from me" will receive it. Its IPC header address is in ?PIPC and the father establishes the format of this message. When CLI is the father, the message contains an edited version of the original CLI command, the command tree, and CLI file information. The system call ?GTMES is used to examine this message (see chapter 8).

System calls ?RETURN and ?TERM effect process terminations. In either case, a message is sent to the father process. The default IPC header layout is :

```
?ISFL   0
?IUFL   term code/PID
?IDPH   0/dad's PID
?IDPL   0
?IDPN   ?SPTM
?ILTH   0
?IPTR   0
```

The termination codes in ?IUFL are defined as follows :

```
0   normal termination (self termination)
1   user trap
2   console interrupt (^C^B or ^C^E)
3   terminated by father
4   terminated by system (fatal error)
```

Optionally, this default header can be overridden by ?RETURN or ?TERM. See the "AOS Programmer's Reference Manual" for details on these system calls. Details on the format of the termination message can be found there too.

While AOS is processing a termination, the system module PRCNG invokes ISEND/SIPC to send the termination message to the father process. However, this is not sent if the father is the root process or is also terminating.

7.2.12 GHOST IPC

When a user's ghost is initialized, a port is assigned for each of his tasks. Tasks 1,2,3,etc... are assigned ports 200,201, 202,etc... One use of these ports is for communication with the console while the debugger is running. (The debugger runs in the ghost.)

If the ghost issues ?GPROC, he has the option of sending a GHOST-to-GHOST message on port 200 . If AC0<>0, then AC1 points to a message which is sent during process creation, invoking SIPC in ISEND. The message consists of a list of the generic file names, so that the new ghost will be able to open generic files.

Format of the GHOST-to-GHOST header :

```
?ISFL   0
?IUFL   0
?IDPH   0/PID
?IDPL   200
?IOPN   0
?ILTH   AC0
?IPTR   AC1
```

7.2.13 IPC notes

a) All above described mechanisms part of the IPC obey the "IPC lock discipline" : while any process accesses the spool file or JORR chain of process R, the spool file CCB of process R is locked.

b) Take advantage of the IPC direct calls implementation and avoid disk I/O overhead when using IPC by keeping an IREC up. By keeping an IREC up, when the ISEND comes home, a match is determined by looking at the outstanding receive chain, a chain running off the receiver's process table and through system virtual buffer space ("GVMEM space"). This data is accessible by remapping operations, no I/O is involved. However, if the matching IREC is not up, the sender's message must be spooled. This necessitates restarting the call as non-direct, bringing in may be two other overlays ISEN1 and ISEN2, and then accessing the spool file. Even if there is a zero-length message, we still need to write to the spool file to add the entry to the spool file directory.

c) Along with keeping an IREC up, also keep the spool file empty. When an IREC is issued, the receiver must check his spool file to see if his match is there. If the spool file is empty, a flag in its CCB will indicate so, and the spool file itself will not be accessed. Otherwise, the call must be restarted, another overlay (IREC) possibly read in, and the spool file directory at least must be brought into core to examine for possible matches.

d) IPC privilege is dangerous! For example, if a sick user sends many (or large) messages to JO's ports which she is not receiving from, JO's spool file will explode with "File space exhausted". Even if one small message is sent and remains in JO's spool file, JO will become sluggish when doing IPC.

e) IPC privilege is fatally dangerous. Sending extraneous IPC to the PMGR will cause a fatal 12.

f) IPC is not the right tool for fast synchronization. With both processes resident, interprocess communication using the above described mechanisms requires about 8 to 12 milliseconds to complete...

7.3 The Connection Manager

All connection management system calls are implemented in overlay CONX, which has the following entry points :

SERVE	user entry for becoming a server
RESIGN	user entry for resigning as a server
CONX	user entry for becoming a customer of a specified server
ICNCT	system entry for establishing a connection
DCONX	user entry for breaking a connection
PCONX	user entry for passing a connection from one server to another
VCNCT	user/system entry for verifying a connection
MBTC	user entry for moving bytes to customer
MBFC	user entry for moving bytes from customer
TBC	system entry for breaking a connection on a TERM or a CHAIN

7.3.1 The Connection Table

The Connection Manager maintains a connection table, CNXTB, for which a quarter page address is defined in STABLE. The quarter page for this table is allocated during the first ?CON establishing a connection between the first server and its first customer. Any subsequent ?CON will build a new entry into the connection table, up to 127. entries maximum.

The format of the connection table is as follows :

```
Word 0: CXLNK   reserved (will be a link word for extending
                the table)
Word 1: CXSIZ   number of entries, i.e. of active connections
Word 2: CXENT   offset of first entry
```

An entry is two words long :

```
CXPID   PID couplet (customer/server)
CXSTS   status of this connection
```

Two status bits and one flag bit are presently defined :

```
CXBMC = 1B0     connection broken by customer
CXBMS = 1B15    connection broken by server

?COBIT = 1B1    do not send an obituary message to customer
```

7.3.2 SERVE / RESIGN logic

SEPVE simply sets bit BPSRV in flag word PFLG4 of the caller's process table and returns.

RESIGN checks if the caller is a server, turns off bit BPRSV in the caller's process table, maps the connection table to slot 66000 and sets bit CXBMS in the status word of every entry with this server's PID. If the obituary flag is not set for this particular connection, an IPC header is built using ?SPTM as origin port and 0 as local destination port, and an IPC termination message is sent via SIPC (remember, this entry skips privilege checks).

7.3.3 CONX / ICNCT logic

CONX retrieves the server's PID associated with the name supplied by the caller if necessary, and maps the server process table via GPTBL. Checks are performed to insure the server invoked has bit BPSRV set and the caller does not try to connect to itself.

If the connection table does not already exist, a quarter page of memory is requested and its address stored at CNXTB in page zero. The connection table is mapped to slot 66 and scanned, looking for the appropriate customer PID / server PID couplet. If found we return, if not found we allocate a new entry for this couplet and set the obituary flag in the status word according to the caller's input.

Note that the state of the obituary flag cannot be reset without breaking the connection. A second ?CON with same customer and server will not change it, whatever input is in AC1.

ICNCT is the internal connect entry point and shares the same code path as CONX. The input AC0 is different, its format being already a customer PID / server PID couplet. This entry is called from PROC3 and SRI.

7.3.4 DCONX logic

DCONX builds a couplet using caller's PID and target PID, the connection table is mapped to slot 66000 and scanned looking for this couplet. If the entry does not exist the couplet is reversed and the table scanned again. When the entry is found, we check and/or set the appropriate "Broken Connection" status bits. If the result of the call is that both bits CXBPC and CXBPS are set, the entry is cleared. If the result is that only one of the bits is set, a break message is sent to the other party; this independently of the state of flag ?COBIT. The IPC header for the break message is as described above.

7.3.5 PCONX logic

PCONX fetches the target process table and checks the server bit. The caller's PID is compared to the target PID as they must be different. The connection table is then mapped to slot 66000 and scanned for a couplet of the form customer's PID/caller's PID. If the connection is found and neither of the "Broken Connection" status bits is set, the new server's PID is stored into the right byte of the entry.

7.3.6 VCNCT logic

This entry simply verifies the existence of a connection by mapping the connection table to slot 66000, scanning it looking for the right couplet, and if found checking the status bits to verify that the connection is not broken.

7.3.7 MBTC / MBFC logic

The code paths for both calls are basically the same. First the packet is moved to the stack and the customer's process table is fetched. The caller's server flag bit is checked, the connection table is mapped, the connection verified and its status checked to make sure it is not broken. Then we chain to the common code for all "Move bytes" system calls, i.e. IMBC in SSOV8.

IMBC checks the byte count for the request, which must be between 0 and 2048, and ensures the target buffer will not overflow. If the customer process is eligible, its extender and map are mapped via MAPP, the buffer space via RDM70 and the bytes effectively moved using routine MTMV8 or MFMV8 in SGSU6. If the customer process is not eligible the bytes are moved to/from the swapfile.

7.3.8 TBC logic

TBC is an internal entry point called from overlay CLNUP on a process termination or chain.

The connection table is mapped to slot 66000 and scanned looking for the presence of the terminating process' PID in a couplet. For every couplet where this PID is found as a server, the "Connection broken by server" bit is set and an obituary message sent to the customer according to the state of ?COBIT. For every couplet where the PID is found as a customer, the "Connection broken by customer" bit is set and an obituary message is always sent to the server.

It seems a check should be performed to see if any entry in the connection table could be cleared due to both "Connection broken" bits being set after this termination. But, as of AOS Rev. 3.03, this is not done.

CHAPTER 6 - GHOST

(updated for AOS Rev. 3.11)

8.1 Introduction

Each user process has two associated address spaces:

- the user (or primary) context
- the ghost (or secondary) context

It might be appropriate to discuss briefly the concept of the "ghost" process. Most people with experience in other operating systems (including RDOS) are familiar with the concept of system supported user resource request - or system calls. In the great majority of these systems a user makes a request for operating system support by some call mechanism which is "trapped" by the operating system. The operating system will then process the request internally, assigning stacks, buffers, and other resources as needed. In general, these assigned resources are allocated out of a general resource pool, usually with a finite limit, thus building in some sort of intrinsic system bottleneck. In addition, some operating system tuning will be required at sysgen time to provide an adequate supply of these resources for proper system performance.

This issue can be further complicated on systems with a small logical address space by causing the operating system to do additional memory management in order to provide adequate resources (buffers etc.). This overhead can be substantial and fraught with software problems due to its complexity.

This leads us directly to the concept of the "ghost". The ghost process is an elegant technique for providing user specific resources in a separately managed address space. The official name for the ghost context is the secondary context, as opposed to the user or primary context.

When a user context is loaded with a program, the ghost is not immediately initialized if the entire context is unshared. If the user context contains shared areas, the ghost is initialized after the program image is loaded : on an initial program load (in overlay SSOV3), we exit to ghost via IGHOST if a shared area has to be loaded for this process.

In effect the ghost is a separate program consisting of code and data (user file buffers) running in parallel with the user program. Ghost performs such functions as resolving generic filename references, deblocking records to/from block devices, and processing partially (i.e. pre- and/or post-processing) or completely certain system calls made by the primary context. For instance, if a user requests that a 100. byte record be read from a file, the ghost will process that request by doing a direct block read of 512. bytes into a ghost buffer, unpack the record and move the requested 100. bytes into the user's record buffer. The AOS kernel sees only the direct block request into the ghost's address space. This is exactly the same when the user makes a direct block request himself. The ghost now does the work of unblocking the record and moving it into the user's primary context record buffer.

The ghost is considered an 'alter ego' of the primary context as opposed to another service module process such as PMGR. This is due to the tight coupling between the primary and secondary contexts. For example, the ghost will use the TCB of the task in the primary context that did the ghost call, saving the contents of the TCB for itself first and setting a bit indicating that task is running in the ghost mode, so that the system scheduler may set the user's map to reflect the secondary context. In addition, some user specific dirty areas are maintained in the ghost as well as the symbolic debugger. In this light, the ghost is just an extension of the user, running in parallel. One of the benefits of this approach is to allow the user to be more accurately charged for his use of system resources, a large part of these resources being actually in his ghost.

8.2 Interfaces

8.2.1 User - ghost interface

The ghost receives control whenever the user makes a ghost call. The ghost call is treated initially just like a regular system call : SCALL determines that the call is a ghost call, and executes the appropriate SYC instruction. If this is the first ghost call, SCALL triggers ghost initialization. The system receives control and knows from the type of SYC that this is a ghost call. It then passes control directly to the ghost, which turns on the "in ghost" bit in the user's TCB. When the ghost is done processing, it turns off the "in ghost" bit, and executes an SVC with ?SCGS in AC0 (?SCGS is "ghost reschedule" code, defined in PARU). SVC as well as SCL can be used to issue an explicit call to AOS (see chapter 6). This causes control to pass to the scheduler, which determines when the user will next be scheduled.

Data interfaces between the ghost and the user have two requirements. When the user is making a ghost call, the ghost will need to read and write AC's, system call packets, and associated data buffers in the user's space. This data gets to the ghost from the user and back again under the control of the ghost, via ?MBFG and ?MBTG calls. These are system calls which move bytes to the ghost from the user and vice versa. The ghost will also need to access the user's TCB's : it does this by mapping the user's page zero into its own context.

8.2.2 Ghost - system interfaces

The ghost interfaces to the system in nearly the same way as the user, although it does not have an SCALL bound into it and does its own setting up of SYC instructions. The system distinguishes between the user and ghost by reading the "in ghost" bit in the user TCB's. Some system calls made by the ghost may require the system to access certain parameters in the user primary context. If so, flags indicating this are passed, usually in the AC's. Other than these differences, the system calls are processed as if they were for a primary context.

The system supports the ghost with the documented system calls such as ?RDB, ?SPAGE etc. In addition, there are some ghost specific (and undocumented) calls to perform such operations as:

?MBFG	move bytes from ghost
?MBTG	move bytes to ghost
?ABTC	abort a system call

etc. For a complete list, see SYSID.SR

8.2.3 Ghost - PMGR interface

When a user issues ?READ/?WRITE to a programmed I/O device, the ghost sends an IPC message to PMGR. PMGR has 256. device ports, of which consoles use three (read/write/control) and most other devices two (e.g. write/control for a line printer). PMGR initially just gets a header from the ghost, and if data movement is required, he issues ?MBTU or ?MBFU to move the bytes to/from the user primary context.

The ghost uses local IPC ports 200 to 377 for PMGR, EXEC, and any other user interfacing.

8.3 Ghost structure

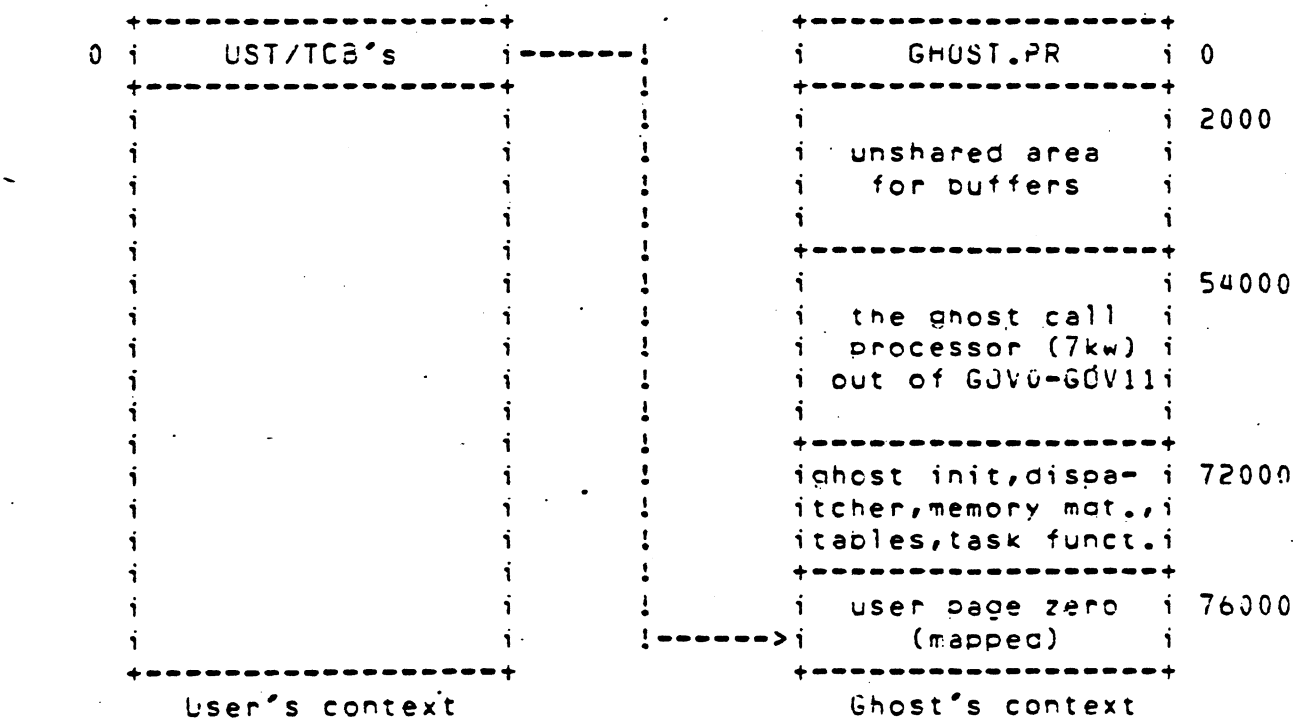
The ghost consists of an overlay manager, a dispatcher, memory management routines, and general purpose subroutines. The functions visible to the user exist in "libraries" of overlays. Currently two such libraries exist:

GHOST.OL — contains I/O and other ghost functions.

DEBUG.OL — contains the user debugger, DEDIT, and the system dump analyzer SYSDMP.

The ghost and its libraries are shareable and write protected, and the overlays are totally relocatable (contain no address constants) since they are dynamically loaded for execution. The unshared code contains the library table, entry table(s) and descriptors, buffers, stacks and other process relevant data. Each process has its own ghost, and the execution of one process ghost is totally unknown to any other ghost. Each process may have multiple tasks using its ghost at the same time. Please remember that there are no "ghost" or "secondary" tasks as such. Tasks exist in terms of the process and may run in either the ghost context or the primary context.

The logical/physical organization of the ghost can be represented as follows :



- GHOST.PR is loaded at ghost initialization time and then frees most of itself to the unshared area for buffers.

- GHOST.OL contains all ghost processing modules using shared overlays. Any single overlay is 2000 words long, and a .BLK is used as last instruction in an overlay, which forces alignment to what will be a memory page boundary.

- The ghost processes system calls for the user by mapping the user page zero (UST/TCB's) into its page 31. (76000).

- The ghost has no TCB's of its own. A user TCB logically exists in both the user's and the ghost's context, and may execute code from either context. However, in order to switch contexts, the TCB must go through a system call (SCL). When the ghost needs a TCB to keep track of itself, it copies the user TCB into a scratch area inside ghost space, inserting the appropriate information into the user TCB.

8.3.1 Ghost boot and initialization

Ghost initialization is a two phase process. The first time a process makes a ghost call AOS will set up the ghost. This is done by mapping page zero (containing all the TCB's) of the primary context to page 37 (logical addresses 76000 - 77777) of the ghost. The ghost bootstrap GHOST.PR is then loaded into page zero of the ghost.

GBOOT in GHOST.PR saves the current user's PC and carry. It then allocates space for temporary storage of the TCB and copies the active TCB information into this area, thereby saving the state of the caller. GBOOT then requests some shared pages and opens GHOST.OL - or DEBUG.OL if the debugger was requested - for shared usage. It determines where in GHOST.OL the GINIT/GHOST dispatcher overlay is located, in the same way essentially as a ?OVLD would - by looking at the .ENTO information in the overlay table -, and the dispatcher is loaded via an ?SPAGE. Of course, if GHOST code is currently in memory for another process, only a remap is necessary to share it. Control is now passed to the GINIT code.

GINIT's job is to set up the ghost environment (much like the FORTRAN runtime initializer). This consists of determining how many tasks can be active, from the user's UST located in page 37 of the ghost, and computing the size needed for extender areas, the channel definition tables and the library tables. If the present ?NMAX for ghost is not high enough, a ?MEMI is executed to request more memory. If this ?MEMI fails for some reason, for example if you have opened more files than the ghost can handle buffers for, the process will be terminated with a ghost error.

After space has been acquired, GBOOT will allocate an extender for each TCB and put the user's TCB information into the appropriate extender for the currently active task. Above the extender area a stack will be allocated for the current TCB. The channel definition tables (CDT) and library tables are set up and initialized to all 1's. If there is any remaining room, the left over memory is allocated into powers of two to the memory buffer pool chains for use later on by normal ghost processing.

Finally, the ghost dispatcher address is placed in the UST for proper scheduling, all user TCB's which were marked as waiting for a ghost initialization are cleaned up and the initializer exits into the AOS kernel with a request to reschedule the ghost through the dispatcher.

One thing to keep in mind is that the ghost is the user's secondary context and it utilizes the TCB of the task requesting ghost service as the TCB for its currently executing code path. This is why an extender area is set up for each user TCB to save the user task state when operating in the ghost context.

8.3.2 Ghost memory management

This module is called GHMIS and contains the memory management facility that is local to the ghost context. These routines provide the following services:

- get/release a shared overlay area
- manage buffer pools
- dequeue/enqueue an area to the free list
- get and send words or bytes into or out of the ghost work areas, using ?MBTG/?MBFG kernel calls
- get stacks
- search strings

The ghost method of memory management ("buddy system") is the same - slightly modified - as used by the AOS kernel for the management of AOS 'GSMEM' dynamic memory buffer pool in AOS logical address space. The free memory areas are put on ten memory chains of 8 to 4096 word length buffers. The chain heads are locations in ghost page zero.

8.3.3 Ghost task handler

This module is called GTASK and it is the ghost system call logic code. Its functionality is somewhat similar to that of 'SCALL' in the user's context. These routines provide for the handling of the ghost requests to the AOS kernel by saving its state in a TCB (in user space) and executing a SVC instruction to enter the kernel. There is a second entry point, 'SCHED' which allows for a ghost reschedule request to the AOS kernel. In addition, this module contains utility routines for managing the ghost's stacks, locking and freeing code paths, finding and readying a TCB waiting on a key, and performing inter-overlay calls with virtual returns.

8.3.4. Ghost overlays

The main functionality of the ghost as far as a user is concerned is provided by the ghost overlays GOV0 to GOV11. These modules implement the ghost calls themselves. The logic of how specific user requests are processed is relatively straightforward : parse the request, build a packet, make a system call if necessary, and return the information to the user. The details can be derived by inspection of the particular module, and there is very little interaction with other modules.

Those who need more information should start by looking at 'GDICT', the ghost dictionary, to determine which ghost overlay to decode which implements a particular ghost call.

8.4 Ghost calls

AOS system call processing is handled on a dual level. Basically, all user instigated system calls (as opposed to AOS internally generated system calls handled by the deamon mechanism and not covered here) are decoded as either "basic" calls or "ghost" calls.

In a general sense, "basic" calls are those which are handed directly to the AOS kernel for immediate processing. "Ghost" calls are those which require either pre-processing by the GHOST context prior to being handled by either the system or the peripheral manager, or post-processing by the GHOST after such handling. Once SCALL has determined what type of call is being made, the request is routed either to the GHOST or to the AOS kernel.

Within the ghost calls we can distinguish the following categories :

- 1B0 in ?TSYS of a user TCB indicates that this system call is initially processed by the ghost. (See SYSID.SR).
  For example :
  If user issues ?OPEN, ghost issues ?GOPEN
  ?CLOSE ?GCLOSE
  ?READ ?RDB, or ?ISEND to PMGR
  ?WRITE ?WRB, or ?ISEND to PMGR
  ?CHAIN ?GCHAN
  ?RETURN ?TRMPR
  ?TERM ?TRMPR
  ?PROC ?GPROC

- System calls fully processed by ghost; these calls may interact with PMGR.
  ?GPOS, ?SPOS, ?SEND, ?ENQUE, ?EXEC, ?GCHR, ?SCHR, ?ASSIGN, ?DEASSIGN, ?GTMES, ?ERMSG, ?OVOPN, ?CTUD, ?CDAY, etc.

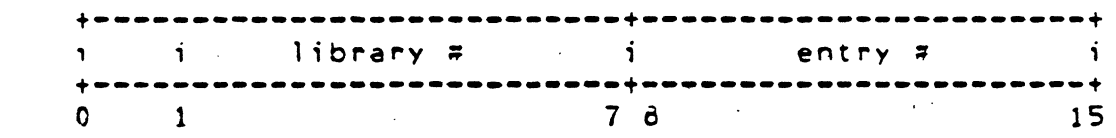
- System calls issued by ghost
  ?GABORT abort ghost
  ?GTEST enter ghost at dispatcher test entry point

- System call issued only by user's SCALL module
  ?IGHOST initialize the ghost

8.4.1 Processing a ghost call

When a ghost function is requested by a program, the assembler sets up two words of code. The first word is a JSR to the system call processor, the second is a gate word designating the desired function.

The gate word is set up as follows:



Bit 0 is a flag bit. If bit 0 = 1 => this is a ghost call
bit 0 = 0 => this is a system call

Bits 1-7 are the library number. Currently only 2 libraries are defined :

0 ghost functions (i/o, return, etc.)
1 debugger (user debugger)
2-127 are reserved for future use.

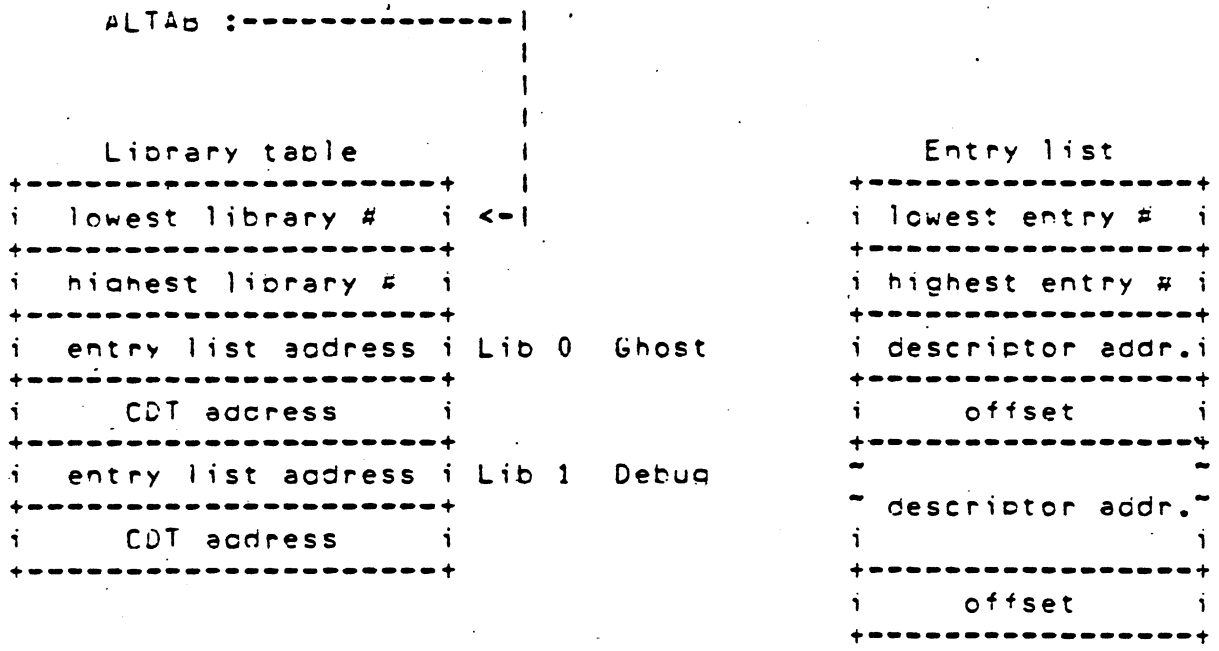
Bits 8-15 are an entry number into the library for the particular call.

The ghost and debugger libraries are made up of a library dictionary and overlays that contain the program code. The dictionary coordinates the entry numbers with the overlays that contain the paths to process the function and the starting address of the code within the overlay. This data is set up at bind time.

The binder loads the system call processor SCALL into the user's program. When a system call is made, SCALL determines whether the call is a system call or a ghost call by checking bit zero of the gate word. Control is sent to the system using SCL or SVC, and AOS gives control to the ghost if it is a ghost call. Of course, control stays in the system if it is a system call.

Once a task transfers to the ghost, the ghost saves the state of the task by copying the task state into the task 'extender'. One extender is allocated for each task. The task must then determine which overlay it wants, using the following look up procedure.

The ghost checks the library number against the library table. The library table is pointed to by word 'ALTAB' in ghost page zero. The first two words of the library table are the lowest and highest legal library numbers respectively. Each library has a two word entry : the first word is the address of an entry list for the library, and the second word the address of a table describing the library file. This table is called a channel definition table or CDT.



The entry list starts with two words that contain the legal entry number range. The first word is the lowest legal entry number and the second word is the highest legal entry number. Each entry has two words associated with it. These words are set up sequentially by entry number after the low-high limit words. For each entry the first word is the address of the descriptor (see below) and the second word is the starting address, relative to the beginning of the overlay for the specific call. Each ghost call has its own entry number.

One descriptor exists for each overlay. Since an overlay may have more than one entry, many entry numbers in the entry list may point to the same descriptor, although they will have a different offset in the same overlay.

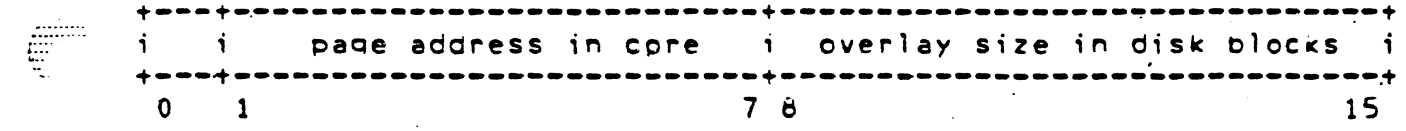
The entry list and associated descriptors are built dynamically at runtime. When a request for a specific library routine is made, the state of the entries is checked. If non-existent, memory is allocated for an entry list and then the dictionary is read into shared space.

The dictionary consists of multiple three word entries, in order by routine number and describing the location of the routine in the overlay file in terms of block number and entry point as an offset from the beginning of the said overlay. From this dictionary information, the entry list and descriptors are created.

The descriptor has the following format:

Offset word in descriptor	Contents
DSFLN	least recently used forward link
DSBLN	least recently used backward link
DSOFS	status, address and size info (see below)
DSUSC	status and use count (see below)
DSCDT	address of the library channel definition table which the library file is opened on
DSSBK	overlay's logical disk block address in library file

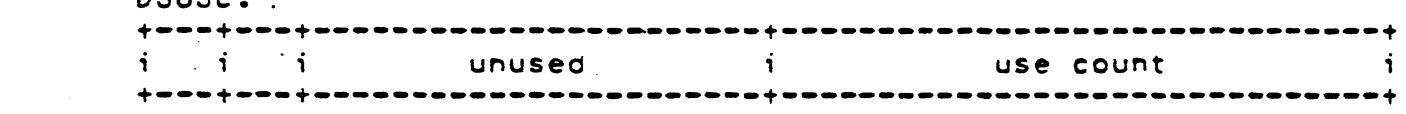
DSOFS:



bit 0 always = 1

"Page address in core" = bits 1 to 7 of the logical memory address where the overlay has been loaded.

DSUSC:



bit 0 = 0 => not loading
bit 0 = 1 => loading
bit 1 = 1 => locked in core

use count = the number of active paths in the overlay.

The descriptor is checked for state of the overlay. The use count is incremented. If the overlay is not in memory, it is loaded and the page address in DSOFS filled in. The LRU (least recently used) chain is updated and execution proceeds at the offset in the overlay indicated by the entry list.

Every ghost call has two returns. The first word after the call is the error return with error code in AC0, the second word is the normal return, with values passed in AC0 - AC2 depending on the call. The ghost initially sets up the return for the normal return and if the ghost routine incurs an error it decrements the return address and places the error code in AC0 on the stack.

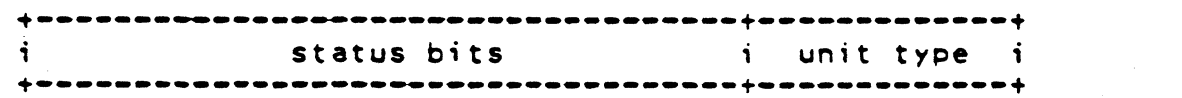
8.4.2 Example

Here is an example of ghost processing a user system call. The user issues ?OPEN, and after some pre-processing its ghost issues a ?GOPEN with these inputs :

Input AC0 : byte pointer to pathname
AC1 : channel number (if -1, AOS will assign)
AC2 : B0 is context indicator for packet
B0 = 1 in user context
B0 = 0 in ghost context
bits 1-15 : pointer to packet

And AOS returns as output :

Output AC0 : device information as follows, this is the CCB status word, offset CBSTS of the CCB (see PARFS).



Status bits
B0: non-disk unit type CCB
B1: error occurred
B2: unpend task on unlock
B3: CCB lock
B4: restart request
B5: file is shared
B6: command bit 2
B7: command bit 1
B8: enable VFU (line printer)
overrice LEOT (magtape)
B9: I/O is illegal
B10: message in file (IPC)
inhibit initial FF (line printer)
file number set (mag tape)

Unit type
0 - Mag tape
1 - MCA
2 - Synch. Line
3 - Dch LPT
4 - Dch LP2

Command bits
00 read
01 write
10 delete
11 read system block (read single block into system buffer)

Output AC1 : the system assigned channel number.

8.5 GTMES processing

The generic filename assignements are maintained by the ghost for the user. In order to resolve the generic filenames for a new process, its father's ghost sends an initial IPC message to AOS (see chapter 4, process creation). The message contains the ?PROC packet that was used to create this process and where the initial generic filename assignements can be found. If the generic filenames are defaulted, the ghost will work back up the process tree to get the generic filenames of the father process. This initial IPC message is spooled to the process 'IPS' spool file in :PROC. Therefore, if the ghost is not initialized when the process is created, the message can still be retrieved at some later time.

The ?GTMES is a ghost call the entry point to the code which implements this call is GTMES in GOV3. The ghost uses ?IREC to receive this message and then decodes it according to the caller's input in the ?GTMES packet. So that the caller can re-issue the ?GTMES, ghost saves the initial message forever in ghost space.

As a flag to indicate whether a previous ?GTMES has been issued, location MSHDR in ghost page zero initially contains a zero. During the first ?GTMES, the IPC header address is placed in this location. Subsequent ?GTMES calls simply access the initial message via the header address in MSHDR.

The first ?GTMES call issues ?IREC to retrieve the initial IPC message from its spool file, with ?ISFL = ?IFSOV + ?IFRFM + ?IFNBK, and ?IPTR = ?ILTH = 0; this ?IREC is performed to see if there is an initial IPC message and to determine its length. Then a second ?IREC is issued with the appropriate length in ?ILTH and a pointer to some adequate ghost buffer space in ?IPTR.

The user can "steal" the initial IPC message from his ghost if he receives this message explicitly before doing a ?GTMES. Also, having a large initial message will effect a large ghost when using the ?GTMES.

8.6 Ghost notes

- When user issues ?READ/?WRITE to block device, ghost allocates a buffer for itself :

Magtape	buffer size = at least one physical record size
Disk	buffer size = multiple of 256. words
Dch LPT	buffer size = 1024. words

  then ghost issues ?RDB/?WRB to AOS. The I/O is done directly to the ghost, which determines what portion of the block should be moved to or from the user's context buffer, using

?MBTG	move bytes from user to ghost
?MBFG	move bytes from ghost to user.

- If user ?READ/?WRITE call is completely device clock aligned, i.e.

  a) The user buffer is word-aligned, and
  b) the record begins on a disk sector boundary, and
  c) the record size is a multiple of 256. words,

  then I/O is direct to user context via IOCAL in AOS. MASM/CLI/BIND use this technique to speed up processing and reduce unshared ghost area usage.

- Ghost uses at least 1Kw unshared memory per user context for a generic filename table.

- The debugger is an additional overlay node in the ghost.

- At normal process termination, all ghost buffers are flushed; however, abnormal process termination might not flush the buffers.

- Ghost code is globally shared and recursive.

- Ghost is variable length buffered and will do multi-sector transfers. But it is not multi-buffered, multi-tasked, or multi-contexted.

- Ghost system call processing time is charged to the user. However, unshared memory is not.

8.7 Labelled tape handling

8.7.1 Introduction

The intent of this section is to familiarize the reader with AOS processing of labelled tapes and some problems which may be encountered when we attempt to exchange tapes among non-AOS installations. The various labels and the fields within each label are described here. They appear in the same order as they would on the tape. Each label is recorded in a separate block and is eighty characters long.

As of AOS Rev. 3.11, labelled tapes handling spans over the following ghost modules :

- GDV6 labelled tape open, called from regular open code when file type is determined to be labelled mag tape; close; routines to search for matching beginning of fiel set, to remount a volume, to read/write header/ trailer labels; labelled tape error processing.

- GDV7 conversion routines for labels; dates, retention period, expiration date, ASCII - binary conversion, record format conversion.

- GDV11 file/device close; read/write block; routines to compare strings with label contents, move one field to another, release memory, write a label; ?LABEL processing.

Besides these ghost modules we have the tape labelling utility LABEL.PR, short straightforward program which just issues a series of system calls, ?LABEL in particular.

8.7.2 Label Field Definitions

Volume-Header Label (VOL1)

Pos	Field Name	L	Content
1-3	Label Identifier	3	VOL
4	Label Number	1	1
5-10	Volume Identifier	6	Assigned permanently by the owner to identify this volume.
11	Accessibility	1	Indicates restrictions on access to information in the volume.
12-37	Reserved	26	Spaces.
38-51	Owner Identifier	14	Identifies owner of the volume. Not used by AOS
52-79	Reserved	28	Spaces.
80	Label-Standard Version	1	Indicates the version of this standard to which the labels and data on this volume conform.

First File-Header Label (HDR1)

1-3	Label Identifier	3	HDR
4	Label Number	1	1
5-21	File Identifier	17	File name.
22-27	File-set Identifier	6	Identifies this file set among other file sets.
28-31	File Section Number	4	Identifies this section among the sections of this file.
32-35	File Sequence Number	4	Identifies this file among files of this file set.
36-39	Generation Number	4	Distinguishes among successive generations of this file.
40-41	Generation Version Number	2	Distinguishes among successive iterations of the same generation.
42-47	Creation Date	6	Date of File Creation.
48-53	Expiration Date	6	Date on which this file may be overwritten.

54	Accessibility	1	Indicates restrictions on access to this file.
55-60	Block Count	6	000000
61-73	System Code	13	Identifies the system that recorded this file. Not used by AOS.
74-80	Reserved	7	Spaces.

ex
0 000 AOS 003012

Second File-Header Label (HDR2)

1-3	Label Identifier	3	HDR
4	Label Number	1	2
5	Record Format	1	F = fixed D = variable S = spanned U don't
6-10	Block Length	5	Maximum number of characters per block.
11-15	Record length	5	Specifies the record length in conjunction with Record Format.
16-50	Reserved for System Use	35	Not intended for use in an interchange environment.
51-52	Buffer-Offset Length	2	The Length in characters of any additional field inserted before the first record in a data block. Not used by AOS.
53-80	Reserved	28	Spaces.

First End-of-Volume Label (EOV1)

1-3	Label Identifier	3	EOV
4	Label Number	1	1
5-54	Same as corresponding fields in HDR1	50	Same as corresponding fields in HDR1.
55-60	Block Count	6	Denotes the number of data blocks since the preceding Beginning-of-File-Section Label Group.
61-80	Same as corresponding fields in HDR1	20	Same as corresponding fields in HDR1.

Second End-of-Volume Label (EOV2)

1-3	Label Identifier	3	EOV
4	Label Number	1	2
5-80	Same as corresponding fields in HDR2	76	Same as corresponding fields in HDR2.

First End-of-File Label (EOF1)

1-3	Label Identifier	3	EOF
4	Label Number	1	1
5-54	Same as corresponding fields in HDR1	50	Same as corresponding fields in HDR1.
55-60	Block Count	6	Denotes the number of data blocks since the preceding Beginning-of-File Section Label Group.
61-80	Same as corresponding fields in HDR1	20	Same as corresponding fields in HDR1.

Second End-of-File Label (EOF2)

1-3	Label Identifier	3	EOF
4	Label Number	1	2
5-80	Same as corresponding fields in HDR2	76	Same as corresponding fields in HDR2.

8.7.2.1 Sample Configurations

Various configurations of files that may be produced by AOS are illustrated below. * means tape mark.

Single File, Single Volume

```
VOL1 HDR1 HDR2 *--File A--* EOF1 EOF2 **
```

Single File, Multi-Volume

```
VOL1 HDR1 HDR2 *--First part of File A--* EOV1 EOV2 **
VOL1 HDR1 HDR2 *--Second part of File A--* EOF1 EOF2 **
```

Multi-File, Single Volume

```
VOL1 HDR1 HDR2 *--File A--* EOF1 EOF2 * HDR1 HDR2 *--File B--* EOF1
EOF2
```

Multi-File, Multi-Volume

```
VOl1 HDR1 HDR2 *-File A-* EOF1 EOF2 * HDR1 HDR2 *-First  part File
B-* EOV1 EOV2

VOL1 HDR1 HDR2 *-Last part File B-* EOF1 EOF2 *  HDR1  HDR2 *-File
C-* EOF1 EOF2
```

USER DEFINED LABELS

In addition to the above labels, there also exist User Defined Labels. If User Volume Labels (UVL1-UVL9) are used, they immediately follow the Volume-Header Label (VOL1). AOS does not support User Volume Labels, although their presence should not affect our processing of the tape. These labels are for installation use only, not intended for use in an interchange environment. If User Header Labels (UHLa) are used, they immediately follow the HDR2 label. If User Trailer Labels are used, they immediately follow EOV1 and EOF2, as appropriate. Both User Header and Trailer Labels are supported by AOS, as the user can specify these labels in the extension to the ?OPEN packet.

These labels contain information which is relevant only to the user. They are simply read and written to and from the user's context and are not processed otherwise.

8.7.2.2 Further Definitions

Further information about some of the more obscure fields is given below.

VOL1

Volume Identifier - Should not be duplicated within an installation, although we have no mechanism to guarantee this.

Accessibility - A space means no access protection. Nonspace means additional qualification is required for access to the rest of the volume. As a minimum, an operating system distinguisnes space and nonspace, and should not permit further volume access for nonspace.

Owner Identifier - If this field is all spaces, then User Volume Labels should not be used. If it is not all spaces, then there is at least one User Volume Label. This field is ignored by AOS.

Label-Standard Version - This field was 1 for the 68 ANSI spec and is defined to be 3 for the 78 spec. The ghost currently only processes tapes with a 3 here.

HDR1

File-Set Identifier - This field is the same for all files within a file set. It is assigned by the ghost to be the Volume Identifier of the first volume of the file set. It is propagated by the system.

File-Section Number - File Section Number of the first section of a file is 0001. This number is increased by 1 for each successive volume of the file. It is maintained by the system.

File Sequence Number - File sequence Number of the first file in a file set is 0001. This number is increased by 1 for each successive file of the set. It is maintained by the system.

Generation Number - This value may be supplied by the user or defaulted to 0001. The Generation Number indicates how far removed the file is from the original generation. When opening a file, the ghost will search through the HDR1 labels looking for a file name match. If the user specifies a value for this field in the extended packet, then this value is compared with the one in the label information. If the two are not equal, then an error is returned to the user. This is incorrect, as the intention should be for the ghost to continue looking for a matching filename and a matching Generation Number.

Generation Version Number - This value may be supplied by the user or defaulted to 00. The version number indicates how many times the associated generation has been replaced. Only the most recent version of a specific generation is retained.

Accessibility - This field has the same meaning as the one in the VOL1 label, only it applies to individual files.

System Code - System Code is a constant for a given system. It is inserted by the system that created the file, to identify itself. Not used by AOS.

8.7.3 AOS Processing of Label Tapes

In creating a new label tape, the tape is first prepared using the LABEL utility. LABEL writes a VOL1, HDR1, and EOF1 label onto the tape. This skeleton framework is necessary in order for the tape to be processed. The /I switch is used for creating IBM tapes, it will cause LABEL to translate the labels to EBCDIC.

A label tape can be referenced in one of two ways. The tape can be explicitly mounted by the user with the CLI 'MOUNT' command or the ?EXEC system call. If this is the case, the tape will remain mounted until the user dismounts the tape or terminates. The tape can also be implicitly mounted by the user when he first opens a file on a labelled tape. In this case, the GHOST will mount and dismount the tape each time the user opens and closes a file.

8.7.3.1 Labeled tape open logic

-close original channel

-insure proper volume mounted (via ?EXEC call)

-open the device

-read the first block (VOL1 label)

  -check for VOL1, proper revision number, and proper VOLUME Identifier if not already checked by EXEC

-read the second block (HDR1 label and HDR2 label if present)

  -check HDR2

  -search for matching file name, mounting subsequent volumes if necessary

-if the file is not found and the create bits are not set, then return an error

-if the file is not found and the create bits are set, or if the file is found and the recreate bit is set, or if the file is found and the create bit is set and the expiration date has been reached, then

-prepare a HDR1 and HDR2 label; values in the user's ?OPEN packet, the extension, and system values in the ghost are merged together to form these labels

-write the HDR1 and HDR2 label and user header labels if present

-if we are not going to recreate the file, then

-check for matching filesets, error if no match

-process HDR1 and HDR2 labels; values from these labels are merged into the user's ?OPEN packet, the extension, and system values in the ghost

-check file section number, error if no match

-if extended packet process creation date, retention date, generation number, and label version number; error if generation number and version number do not coincide with the values specified by the user or the defaults

-read and return user header labels if packet is present

-close the device

-open the specific file

-return the extension packet if necessary

-continue regular open processing

8.7.3.2 ?RDB / ?WRB error processing logic

-if error not ERSPC or EREOF then return error code to user

-if read

-open next file

-if EOF1 then real eof error

-if not EOV1 then bad label error

-close file (see CLOSE)

-mount next volume (via ?EXEC)

-enter the label tape open code (see above)

-return to the regular I/O code and restart the last request

8.7.3.3 Labeled tape close logic

-close user file

-open device

-read user trailer labels if file has not been modified

-prepare and write either EOF1 - EOF2 or EOV1 - EOV2

-write user trailer labels if present

-close device

-tell EXEC (this will result in a dismount and rewind of the tape if the user implicitly mounted it)

8.7.4 ANSI Labeling Levels

The ANSI specification defines four upward compatible levels of tape labeling, each of increasing complexity. They are defined below.

Files	1	Single-file single-volume + single-file multivolume
	2	Single-file single-volume + single-file multivolume + multifile single-volume + multifile multivolume
Labels	1	VOL1 HDR1 EOV1 EOF1
	2	VOL1 HDR1 HDR2 EOV1 EOV2 EOF1 EOF2
Record Format	1	Fixed
	2	Fixed + variable
	3	Fixed + variable + spanned

	Files	Labels	Record Format
Level 1	1	1	1
Level 2	2	1	1
Level 3	2	2	2
Level 4	2	2	3

In a level 1 system, the following label fields are processed:

(1) VOL1
- Label Identifier
- Label Number
- Volume Identifier
- Accessibility
- Label-Standard Version

(2) HDR1, EOV1, EOF1
- Label Identifier
- Label Number
- File Identifier
- File Section Number
- Expiration Date
- Block Count

In a level 2 system, the following label fields are processed: (* indicates an addition from the previous level)

(1) VOL1
- Label Identifier
- Label Number
- Volume Identifier
- Accessibility
- Label-Standard Version

(2) HDR1, EOV1, EOF1
- Label Identifier
- Label Number
- File Identifier
- * File-Set Identifier
- File Section Number
- * File Sequence Number
- Expiration Date
- * Accessibility
- Block Count

In a level 3 system, the following label fields are processed:

(1) VOL1
- Label Identifier
- Label Number
- Volume Identifier
- Accessibility
- Label-Standard Version

(2) HDR1, EOV1, EOF1
- Label Identifier
- Label Number
- File Identifier
- File-Set Identifier
- File Section Number
- File Sequence Number
- * Creation Date
- Expiration Date
- Accessibility
- Block Count

(3) HDR2, EOV2, EOF2 *
- Label Identifier
- Label Number
- Record Format
- Block Length
- Record Length
- Buffer-Offset Length

In a level 4 system, the following label fields are processed:

(1) VOL1
- Label Identifier
- Label Number
- Accessibility
- Label-Standard Version

(2) HDR1, EOV1, EOF1
- Label Identifier
- Label Number
- File Identifier
- File-Set Identifier
- File Section Number
- File Sequence Number
- * Generation Number
- * Generation version Number
- Creation Date
- Expiration Date
- Accessibility
- Block Count

(3) HDR2, EOV2, EOF2
Label Identifier
Label Number
Record Format
Block Length
Record Length
Buffer-Offset Length

At each level, the processing of additional label fields is optional. However, all fields within the given labels contain meaningful information, i.e. they should contain the standard default values. If the accessibility field contain spaces, then that volume or file is accessible to the user. If the contents are not a space, then access should be denied unless the system provides additional controls.

8.7.5 Significant differences between ANSI and IBM labels

VOL1
-IBM does not use the LABEL-STANDARD VERSION field. AOS should not expect to find a value in this field when processing IBM tapes. It currently does.

HDR1/EOV1/EOF1
-No difference.

HDR2/EOV2/EOF2
-Does not contain the Buffer-Offset Length field which is currently ignored by the GHOST anyway. Contains a new field, Block Attribute, which has information about the record format.

8.7.6 IBM Record Formats

IBM defines four record formats : fixed-length (F), variable-length for data in EBCDIC (V), variable length for data to be translated to or from ASCII (D), or undefined-length (U).

F = fixed length
The number of records within a block is constant for every block in the file, with the possible exception of the last block which may be truncated. It is required that the buffer length be an integral multiple of the record length.

If producing an ASCII tape, blocks of records may have block prefixes which can contain any data specified by the user. Currently, AOS is capable of reading and writing this record format provided no block prefix exists. When the Buffer-Offset Length in the HDR2 label is incorporated, we will be able to read, but not write, tapes with block prefixes.

V = variable length

This format provides for variable-length records, variable-length record segments, each of which describes its own characteristics, and variable length blocks of such records or record segments. The first 4 bytes of each block and the first 4 bytes of each record contain control information. The first 2 bytes of the block descriptor word specify the block length. This length can be from 18 to 32,760 bytes. The third and fourth bytes are reserved and must be 0. The first 2 bytes of the record descriptor word contain the length of the record. The third byte of the descriptor contains the segment control code, which specifies the relative position of the segment in the logical record. The segment control code in the rightmost 2 bits of the byte has the following meaning :

00	complete logical record
01	first segement of multisegement record
10	last segment of multisegment record
11	segment of a multisegment record other than the first or last segment

With this format, block length is independent of record length. Again, AOS has no concept of segment control words.

D = variable-length records for ASCII tapes.

Same as the above, except blocks may contain block prefixes.

U = undefined-length records

Each block is treated as a logical record and, when creating ASCII tapes, may contain a block prefix.

8.7.7 AOS versus RDOS/INFOS Processing of Label Tapes

In general, AOS and RDOS/INFOS share the same functionality in the realm of label tapes. There are a few points worth mentioning.

In regard to label fields :

1) RDOS/INFOS allows a user to specify the degree of labeling to be produced for a tape. That is, the user may request ANSI levels 1, 2, or 3, or IBM levels 1 or 2. The levels represent increasing complexity in the label fields and increasing flexibility in the record format of the files and file configurations within the volume, i.e. single file single volume versus multifile multivolume. Since AOS lacks the ability to specify levels, a user will encounter problems if he intends to interchange tapes produced by AOS with an installation which only supports a lower level.

2) RDOS/INFOS has the capability of writing User Volume Labels. It also writes the Owner Identifier and System Code fields and supports the Buffer Offset Length. Also, RDOS/INFOS allows the user to specify an accessibility character, although this is never processed. Like AOS, RDOS/INFOS has no protection mechanism for files residing on labelled tapes.

3) RDOS/INFOS informs the user when physical volumes have been switched and when User Labels have been processed. The user may specify the File Set Identifier and the Sequence Number fields. The user has control over the position of the tape when it is closed. The user may also force an end of volume.

In regard to record formats:

1) The BIG difference is RDOS/INFOS does not allow a record to span blocks on label tapes. RDOS/INFOS defines three record formats applicable for label tapes : fixed, variable, and undefined. The user may optionally specify variable blocks and a pad character. The system defaults to fixed blocks and ASCII null (0) as the pad character. Variable records and blocks each have a two word header specifying the length. When writing fixed blocks, if the last record written to a given block does not entirely fit inside the block, then that block is extended with the pad character to its specified length and the record is written in its entirety to the following block. When writing variable blocks, if the last record does not fit, that block is truncated and the record is written to the next block.

8.7.8 Summary of the Various Record Formats

The possible record formats, which exist outside of AOS, as well as the AOS current record formats are listed below :

1) AOS fixed length records
2) AOS variable length records
3) AOS data sensitive records
4) AOS dynamic records
5) AOS undefined records

6) RDOS fixed length records, fixed blocks
7) RDOS fixed length records, variable blocks
8) RDOS variable length records, fixed blocks
9) RDOS variable length records, variable blocks
10) RDOS undefined records, fixed blocks
11) RDOS undefined records, variable blocks

12) IBM fixed length record, implies fixed blocks
13) IBM variable length record, implies variable blocks and spanned records

14) ANSI fixed length records
15) ANSI variable length records
16) ANSI spanned records

Files produced by both RDOS and IBM may optionally have block prefixes. When the Buffer-Offset Length field in the HDR2 label is incorporated into AOS processing of label tapes, we will be able to properly process files with block offsets.

We can process record types 6, 7, 12, and 14 above provided we operate under the constraint that when the file was created, record length was an integral multiple of the block size. If this is not the case, then type 6 will have pad characters at the end of its blocks, type 7 will have variable length blocks, and in type 12, IBM will not be able to read our tape, as they require record length to be a multiple of block size. Again, type 8 will have pad characters and type 9 will have varying blocks, neither of which AOS can handle. We can process types 10 and 11, as there is no de-blocking involved with undefined records. Type 13 will have varying blocks and spanned record descriptors which cannot currently be processed under AOS. Type 15 can be processed if there is an integral number of records within each block, else the blocks will have pad characters or be of varying length. Type 16 cannot be processed.

Note : IBM is capable of writing 32K byte blocks, while Ghost I/O is still constrained to a maximum block size of 8K byte.

CHAPTER 9 - PMGR, THE PERIPHERALS MANAGER (update for AOS Rev. 3.11)

9.1 PMGR Overview

The program "PMGR.PR" or "IOPMGR.PR" is responsible for all character devices, such as : consoles, paper tape readers/punches, programmed I/O (not data channel) line printers), ALM's, async DCU, IOP, card readers, plotters, etc.

All communication with the PMGR is through IPC data. This document assumes the reader is familiar with IPC's and character I/O. This information can be found in chapters 4 and 6 of the AOS System Programmer's Manual (093-000120).

For every PMGR device there is a control port and a read and/or write port. The control port is found via ?ILKUP or ?GOPEN. The read and/or write ports are returned by the PMGR every time the USER opens the device. Control requests must always be sent to the device's control port. Read requests must always be sent to the read port. Write requests must always be sent to the write port.

All data transfers to and from devices are done through the ring buffers. The interrupt level code controls the device/ring buffer flow. PMGR base level code controls the ring buffer/temp buffer flow. The temp buffers are transferred to the user using ?MBTU/?MBFU when the I/O is complete.

When it assigns ring buffers for a device, the PMGR will issue a ?XLAT system call : this call translates a PMGR logical address into an appropriate lump word which the interrupt level code will use for loading the map.

9.2 PMGR Configurations.

The PMGR supports a DCU or IOP for off-loading processing from the host. The 'configuration' of the PMGR depends on which, if any front end processor is available. The three configurations are summarized below :

Function	Non-DCU	DCU Assisted	IOP Assisted
CON0 Interrupt Level Processing	AOS	AOS	Relayed To IOP
CON1 Interrupt Level Processing	AOS	AOS	IOP
CON2+ Interrupt Level Processing	AOS	DCU	IOP
Other PIO Interrupt Level Processing	AOS	AOS	IOP
Base Level Char. Processing	PMGR	PMGR	IOP
Device/PMGR Buffers, Queues, etc.	PMGR	PMGR	IOP
PMGR/User IPC Buffers	PMGR	PMGR	IOPMGR

Note: On a M600, CON0 is on the host I/O bus, not the IOPs. Therefore, interrupts that occur on CON0 must be passed on to the IOP for processing.

9.3 PMGR - Data Interfaces

In this section it is helpful to distinguish between I/O data and control data. The term control data refers to any data which is used to select actions, while I/O data refers to the character data transferred from the devices to the user (or the reverse) by way of the PMGR.

9.3.1 SYSTEM - PMGR I/O Data Interface

The PMGR functions as a device driver for the character peripherals, and as such has intimate connections with the interrupt world inside AOS.

On a character by character basis, the data coming from a device is read directly into the PMGR's ring buffers. This is possible due to the knowledge that the interrupt world has of the PMGR's databases. The PMGR also receives IPC data from the system on the occasion of a process creation, chain, or termination. These messages are used to determine console assignment, and the PMGR will acknowledge them by issuing the ?SGNL system call.

For output, user data is stored into the PMGR's temporary buffers using the ?MBFU system call. The PMGR base level code in IOCOM then transfers the data to the interrupt world's ring buffers, and starts the transfer. The temporary buffer (max 510. bytes) is typically larger than the ring buffer, and in such a case the PMGR task sets a PIB status bit indicating that the PIB is waiting for the ring buffer to empty at least one character, and then the task blocks the PIB.

The interrupt world outputs the character, updates its ring buffer pointers, and unblocks the PIB. This unblock indicates that ring buffer is ready for additional data. The interrupt world also sets DCUWK in AOS page zero, ensuring that the PMGR will be scheduled during the next pass through the AOS scheduler, i.e. at least once every sub-slice (32. milliseconds). If the PMGR is the currently running process, the PIB will be unblocked when the current task relinquishes control. This linkage between the interrupt world and the PMGR base level code may be delayed by as much as 32. milliseconds.

The interrupt code does not "wait" for the base level on a character by character basis. The only condition that will cause the interrupt world to "wait" is if all the data within the ring buffer has been processed before the base level can "stuff" additional data into the ring buffer.

The PMGR base level code will transfer as many characters as possible into the ring buffer when it gets control, thus "catching up" with the interrupt world. Once the PMGR is running, it will process all unblocking events that have occurred since the last time it was in control. The PMGR then processes any further events that occurred while the PMGR was busy. When the event queue is empty the PMGR relinquishes control to lower priority processes.

The buffer management technique for input is basically the same but in reverse order.

9.3.2 PMGR - USER I/O Data Interface

The PMGR and a GHOST/USER function asynchronously as separate processes who communicate through their data interfaces. The AOS scheduler determines independently for each when it should receive control. The IPC between a User's GHOST and the PMGR provides synchronization to the extent that the User process will be suspended on its GHOST's ?IREC until the PMGR responds. In the case of a read or write, the PMGR transfers the data to the primary context by issuing a ?MBTU call or from the primary context with a ?MBFU call. The data is stored in temporary buffers within the PMGR address space.

9.4 PMGR Control Interfaces

The PMGR has all the normal control interfaces with the system that any process has. It may trap, make system calls, and lose control to the interrupt world. Control is returned to it via rescheduling or via the normal return (to user) from interrupt processing.

The interrupt world can and does modify the contents of the PMGR databases such as task's TCB status, PIB's, Queues, etc. and causes an AOS reschedule to post the status change or control function completion. Any event so posted will cause the interrupt world to set the DCUWK flag, indicating the PMGR wants control the next time that the scheduler is entered, which will be at most 32. milliseconds later.

When the PMGR wishes to post a request to the interrupt world, he sets up the request in the PMGR data structures and signals the interrupt world by explicity executing an I/O instruction which will cause the interrupt world to be entered and the posted request to be found. In some instances the PMGR task will wait for the request completion by suspending itself. In these cases, the interrupt world is responsible for waking up the task by resetting the suspend status bit in the TCB and forcing an AOS reschedule.

9.5 PMGR Basic Task Flow

There are six PMGR tasks. One task is dedicated to the time out processing. The other five tasks are general purpose tasks, and are not dedicated. The main code path in the PMGR tries to keep :

1 task waiting on ?IREC
1 task waiting for interrupt events
3 tasks "doing work" or idle

These task assignments are not static. The tasks rotate duty. For example, if an IPC request comes in, the ?IREC task will try to start another task to replace itself, and then go process the IPC message. If the path must block (waiting for memory, or I/O), it will block the PID and then become the "Interrupt" or ?IREC task if there is not a task already assigned. When an interrupt event occurs, the interrupt task will try starting a task to replace itself, and then go process the interrupt event. The task is now a "doer" task. It will continue to completion or to a condition where it must block. In either case it will try to become the interrupt task or the ?IREC task if possible.

The major flow of the PMGR is contained in the module PMGR.SR. The program flow in pseudo-code looks like this :

```
WHILE NUMBER_PIBS_TO_UNBLOCK <> 0 DO
   BEGIN

      IF OUTPUT_DONE AND (OUTPUT_QUEUE = EMPTY) THEN
         BEGIN
            NUMBER_PIBS_TO_UNBLOCK := NUMBER_PIBS_TO_UNBLOCK - 1;
            RELEASE_MEMORY;
         END;

      IF PIB_STATUS <> IN_USE THEN DEQUEUE_PIB;

      IF INPUT_DONE OR OUTPUT_DONE THEN
         BEGIN
            NUMBER_PIBS_TO_UNBLOCK := NUMBER_PIBS_TO_UNBLOCK - 1;
            START_ANOTHER_TASK;
            PROCESS_INTERRUPT_REQUEST;
         END;

   END;    /* OF WHILE LOOP */

/* WHEN NOTHING LEFT TO DO */

IF OTHER_TASK_WAITING_ON_IREC = FALSE THEN
   BEGIN
      OTHER_TASK_WAITING_ON_IREC := TRUE;
      WAIT_ON_IREC;                          /* IN MODULE CONCOM.SR */
      OTHER_TASK_WAITING_ON_IREC := FALSE;   /*                     */
      START_ANOTHER_TASK;                    /*                     */
      PROCESS_IREC_REQUEST;                  /*                     */
   END;
ELSE IF OTHER_TASK_WAITING_ON_INTERRUPT = FALSE THEN
        BEGIN
           OTHER_TASK_WAITING_ON_INTERRUPT := TRUE;
           WAIT_ON_INTERRUPT;
           OTHER_TASK_WAITING_ON_INTERRUPT := FALSE;
        END;     ELSE KILL_SELF;
```

9.6 PMGR Memory Images

The PMGR can have four distinct memory images. These are :

- Memory Image in Host
- Memory Image in IOP.
- Memory Image of PMGR Residual in M/600 Host (MART)
- Memory Image in DCU

The first three images will be discussed in the following pages. For the DCU image, one can look at the modules DCUC.SR and ALDCU.SR.

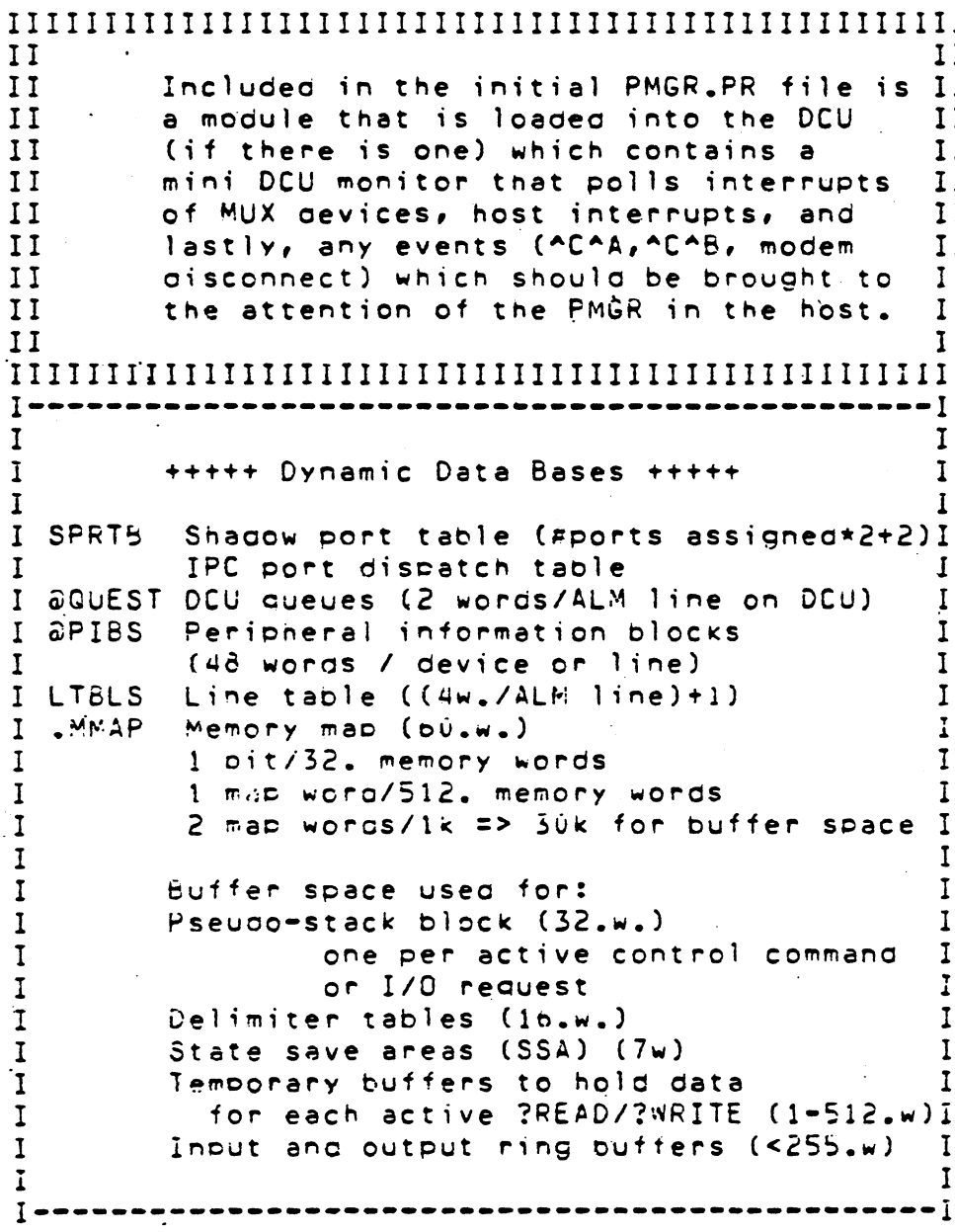
9.6.1 PMGR Memory Image - Non-IOP Host

Module Description	Approximate locations and module names
32K ADDRESS SPACE	
ZREL code	Page Zero
AOS Interface Words:	
NUMB- Number of unblocked PIB's BTCB- Addr of blocked TCB for interrupt world NEVEN- Number of interrupt events waiting	
DCU Interface Words:	
SLTH- Slave to host register DLCK- DCU interprocessor lock HOLK- Host interprocessor lock	
UST	400
6 TCB's 1 Initialization/timer task 5 General purpose tasks for performing whatever function is currently required	423
NREL Code Mainline code consists of logic for allocating the available TCB's to ongoing work governed by the AOS interface words and the necessity to keep that-all-important ?IREC to maintain the incoming work flow. Included are general subroutines for memory management, timeout processing, error recovery, and DCU synchronization	600 PMGR.SR
	2000

This module contains code to receive incoming IPC messages directed at the PMGR's local control ports and to process them according to the following control commands: OPEN Get read/write ports CLOSE ASSIGN DEASSIGN ASSIGN CONSOLE (System only) PROCESS TERM (System only) CHAIN (System only) GET DELIMITER TABLE SET DELIMITER TABLE GET DEVICE CHARACTERISTICS SET DEVICE CHARACTERISTICS SEND A CONSOLE MESSAGE TERMINATE READ REQUESTS SET TIME-OUT CONSTANT GET STATISTICS RESET STATISTICS	2000 CONCOM.SR
This module contains logic to process I/O requests directed at the PMGR's local read and write ports. When the I/O operation is done it does a ?ISEND to the requestor. In a read operation, we are mainly concerned with moving data from each device input ring buffer and placing it into a temporary buffer the size of which will match the particular ?READ in progress. When the proper number of bytes are assembled in the temporary buffer, the data is moved to the user's address space (via ?MBTU) and the read is terminated. In a write operation, the data is moved from the user's address space into a temporary buffer (using ?MBFU). From the temporary buffer, the data is stuffed into the intended device's output ring buffer for subsequent transmission to the device.	5400 IOCOM.SR

This module also contains logic to differentiate between device types for special device-specific functions such as : Non-ANSII conversion of <NL> to <CR><NL> Positioning the cursor before I/O, Upper/lower case I/O or convert to Upper Rubout echo and cursor control,...etc.	11200
This module contains screen-edit logic which is really an extension of the above. It concerns itself with reading the user buffer (using extra ?MBTU), modifying the data according to "line-edit" control characters and returning to module IOCOM when input is completed.	SCEDT.SR
This module contains logic to handle multitasking. The user runtime routines (URT) were probably not written when the PMGR was written, so this module takes the place of URT routines. Also this small subset of URT routines are more efficient than using the more general URT modules. Another routine here is "WAKE" to wake up the ?DELAY task by clearing "waiting for ?XMT" bit whenever timeout processing is active.	12000 TASK.SR
This module contains logic to start input and output so that, henceforth, a device will generate an interrupt at completion of each ouput character or detection of each input character. The routines for fielding these interrupts reside in AOS modules (PERDR.SR, MUXDR.SR, CDRDR.SR). Also included are routines to turn modem lines on/off.	13000 STARTS.SR
	13200

This module handles the internal paths of data between "@PCR-" and @PCL-". Pseudo consoles are internal PMGR data paths between two PIB's. This makes it unnecessary to have actual hardware consoles for software testing.	13200 SUDO.SR
This module supports the card reader. it is an extension of the input part of "IOCOM".	13400 CARD.SR
This module is the Hollerith conversion table needed by the above module. The char set is IBM 029 card punch code. A similar (and less used) module is used for IBM 026 code (HOL26.SR).	13700 HOL29.SR
This module controls the DCU after it is loaded from module DCUI.SR (at init time). The functions involved are: 1- start output 2- modem on 3- modem off in each of the above cases, the command is placed in the Host-to-DCU queue and the DCU is signalled to perform the indicated action.	14000 DCUC.SR
****** Data Bases ****** Mnemonic Description Length(words) ----- Fixed Data Bases ----- DFDT Default delimiter table (16.) NFDT Null default delimiter table (16.) PIDTB PID to PIB's cross reference table (64.) Used to point to the PIB (or PIB chain) associated with every PID known by the PMGR. ----------	14200 INIT.SR
	37000


Included in the initial PMGR.PR file is a module that is loaded into the DCU (if there is one) which contains a mini DCU monitor that polls interrupts of MUX devices, host interrupts, and lastly, any events (^C^A,^C^B, modem disconnect) which should be brought to the attention of the PMGR in the host.
DCUI.SR
+++++ Dynamic Data Bases +++++
Start at approx. 20000
SPRTS Shadow port table (#ports assigned*2+2)
IPC port dispatch table
@QUEST DCU queues (2 words/ALM line on DCU)
@PIBS Peripheral information blocks
(48 words / device or line)
LTBLS Line table ((4w./ALM line)+1)
.MMAP Memory map (bit.w.)
1 bit/32. memory words
1 map word/512. memory words
2 map words/1k => 30k for buffer space
Buffer space used for:
(.MMAP+74(8))
up to 77777
Pseudo-stack block (32.w.)
one per active control command or I/O request
Delimiter tables (16.w.)
State save areas (SSA) (7w)
Temporary buffers to hold data for each active ?READ/?WRITE (1-512.w)
Input and output ring buffers (<255.w)

9.6.2 IOPMGR Image in IOP

Differences between PMGR and IOPMGR are marked thus ***

Module Description	Approximate locations and module names
IOP ADDRESS SPACE (32K)	
	Page Zero
ZREL	
AOS Interface Words:	
NUMB- Number of unblocked PIB's	
BTCB- Addr of blocked TCB for interrupt world	
NEVEN- Number of interrupt events waiting	
DCU interface words: NOT USED	***
SLTH- Slave to host register	
DLCK- DCU interprocessor lock	
HOLK- Host interprocessor lock	
UST	400
6 TCB's	423
1 Initialization/timer task	
5 General use tasks for whatever function being done	
NREL Code	600
Mainline code consists of logic for allocating the available TCB's to ongoing work governed by the AOS interface words and the necessity to keep that-all-important ?IREC to maintain the incoming work flow. Included are general subroutines for memory management, error recovery, timeout processing, and DCU synchronization.	PMGR.SR
	2000

2000

CONCOM.SR

This module contains code to receive incoming IPC messages directed at the PMGR's local "control" ports and to process them according to the following control commands:

OPEN get read/write ports
CLOSE
ASSIGN
DEASSIGN
ASSIGN CONSOLE (System only)
PROCESS TERM (System only)
CHAIN (System only)
GET DELIMITER TABLE
SET DELIMITER TABLE
GET DEVICE CHARACTERISTICS
SET DEVICE CHARACTERISTICS
SEND A CONSOLE MESSAGE
TERMINATE READ REQUESTS
SET TIME-OUT CONSTANT

5400

IOCOM.SR

This module contains logic to process I/O requests directed at the PMGR's local "read" and "write" ports. When the I/O operation is done it does a "?ISEND" to the requestor.

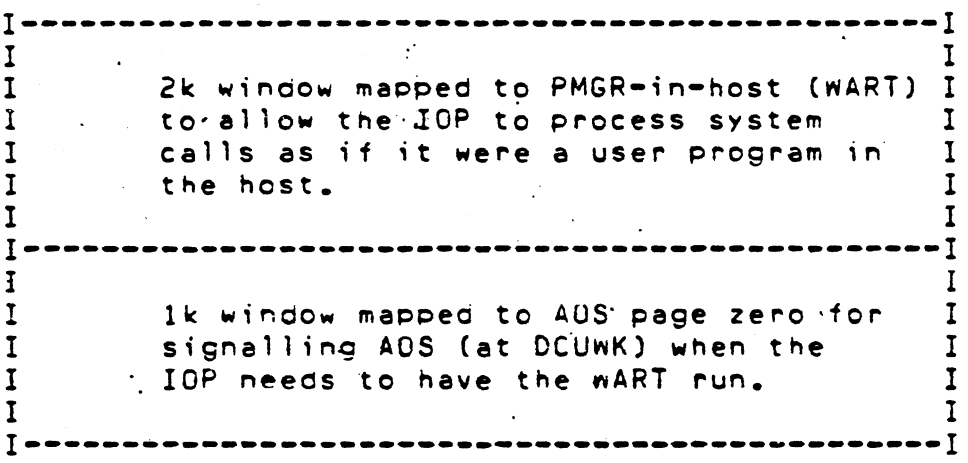
In a read operation, we are mainly concerned with moving data from each devices' input ring buffer and placing it into a temporary buffer the size of which will match the particular "?READ" in progress. When the proper number of bytes are assembled in the temporary buffer, the data is moved to the user's address space (via ?MBTU) and the read is terminated.

In a write operation, the data is moved from the user's address space into a temporary buffer (using ?MBFU). From the temporary buffer, the data is stuffed into the intended device's output ring buffer for subsequent transmission to the device.

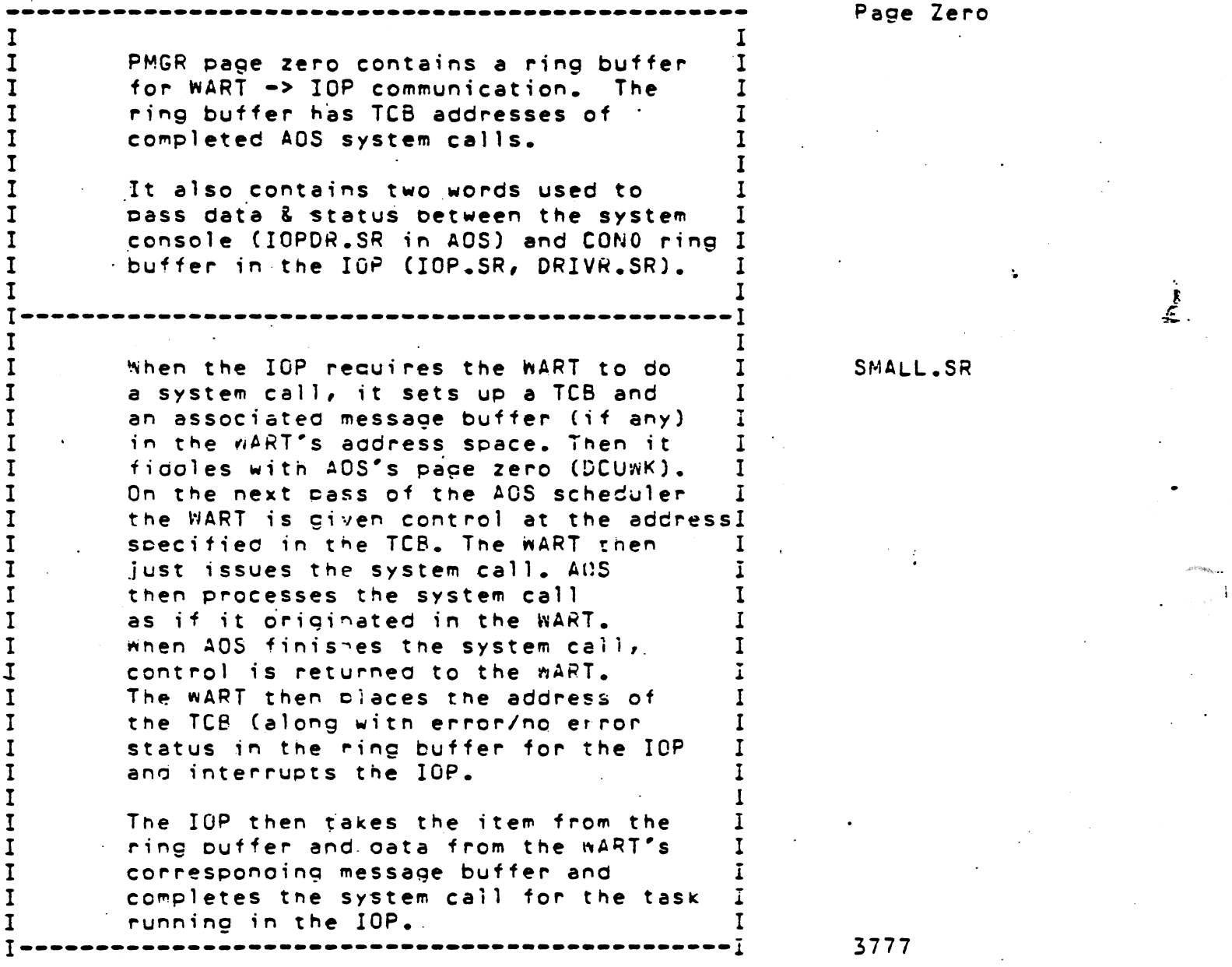
This module also contains logic to differentiate between device types for special device-specific functions such as: Non-ANSII conversion of <NL> to <CR><NL> positioning the cursor before I/O, Upper/lower case I/O or convert to Upper Rubout echo and cursor control,...etc.	
	11200
This module contains screen-edit logic which is really an extension of the above. It concerns itself with reading the user buffer (using extra ?MBTU), modifying the data according to "line-edit" control characters and returning to module IOCOM when input is completed.	SCEDT.SR
	12000
This module contains logic to handle multitasking. The user runtime routines (URT) were probably not written when the PMGR was written, so this module takes the place of URT routines. Also this small subset of URT routines are more efficient than using the more general URT modules. Another routine here is "WAKE" to wake up the ?DELAY task by clearing "waiting for ?XMT" bit whenever timeout processing is active.	TASK.SR
	13000
This module contains logic to start input and output so that, henceforth, a device will generate an interrupt at completion of each ouput character or detection of each input character. The routines for fielding these interrupts reside in AOS modules (PERDR.SR, MUXDR.SR, CDRDR.SR). Also included are routines to turn modem lines on/off.	STARTS.SR
	13200

	13200
This module handles the internal paths of data between "@PCR-" and @PCL-". Pseudo consoles are internal PMGR data paths between two PIB's. This makes it unnecessary to have actual hardware consoles for software testing.	SUDO.SR
	13400
This module supports the card reader. it is an extension of the input part of "IOCOM".	CARD.SR
	13700
This module is the Hollerith conversion table needed by the above module. The char set is IBM 029 card punch code. A similar (and less used) module is used for IBM 026 code (HOL26.SR).	HOL29.SR
	14000
This module replaces DCUC.SR because the DCU code is not used in the IOP. It contains error routines to panic or return error "ERDCV" illegal device if DCU is ever referenced in the IOP.	DCUX.SR ***
	14065
This module is the PMGR's replacement for AOS functions which were supplied to the PMGR in the host. Specifically, the following functions are provided here: System call pre/post processing IOP task management Interrupt processing including Host-IOP intercommunication Special @CON0 interrupt service IOP timer(internal ?DELAY) Device I/O interrupts	IOP.SR ***
	16000
This module contains the interrupt service routines equivalent to AOS modules MUXDR.SR, PERDR.SR, and CDRDR.SR.	DRIVR.SR***

Mnemonic	Description	Length(words)	
	***** Fixed Data Bases *****		
DFDT	Default delimiter table (16.)		PINIT.SR***
NFDT	Null default delimiter table (16.)		
PIDTB	PID to PIB's cross reference table (64.) Used to point to the PIB (or PIB chain) associated with every PID known by the PMGR.		
	+++++ Dynamic Data Bases +++++		Start at approx. 20000
SPRTB	Shadow Port Table (#ports assigned*2+2) IPC port dispatch table		built by PINIT & never released
@QUEST	DCU queues (2 words/ALM line on DCU)		
@PIBS	Peripheral information blocks (48./device or line)		
LTBLS	Line table ((4/ALM line)+1)		
.MMAP	Memory map (60.) 1 bit/32. memory words 1 map word/512. memory words 2 map words/1k => 30k for buffer space		Typically (25000)
Free memory space used for:			(25074)
	Pseudo-stack block (32.) one per active control command or I/O request		built and released by memory management routines at runtime
	Delimiter tables (16.)		
	State save areas (SSA) (7)		
	Temporary buffers to hold data for each active ?READ/?WRITE (1-512.)		
	Input and output ring buffers (<255.)		
			72000


72000
2k window mapped to PMGR-in-host (WART) to allow the IOP to process system calls as if it were a user program in the host.
76000
1k window mapped to AOS page zero for signalling AOS (at DCUWK) when the IOP needs to have the WART run.

9.6.3 IMAGE OF RESIDUAL PMGR IN M600 HOST CPU (WART) ***



9.7 PMGR - General Information

- Controls all character I/O devices (@CONX,@LPA,@TRA,@TPA,@CRA,@PLA)

- Can be configured to run with or without
  - DCU
  - IOP
  - Not simultaneously (i.e. No DCU on IOP)

- Created as resident process and fixed as PID 1

- On an unassisted system, PMGR will grow to 32.k words if necessary for dynamic databases (29.k on an IOP system / limited by the # of Map slots on a DCU assisted system.

  - Never releases memory at runtime after initialization

- Has all privileges and runs in LEF disable mode with device access enabled

- Communicates via IPC's to USERs, GHOST, EXEC and System

- Supports modem controlled devices on all flavors of PMGR

- Actual hardware level control of devices take place from both the PMGR process level and the system interrupt world

- Actual data transfer to/from USER is done via ?MBTU/FU calls

- Several special 'System only' control sequences
  - Process creation (ASCON)
  - Process termination (PTERM)
  - Process chain (CHAIN)

- PMGR memory management element granularity is 32. words.
  - Page boundary crossing not allowed for any data base so interrupt world will not have to do multiple maps

- Major databases are:
  - Peripheral Information Block (PIB)
  - Line Table (LTBL)
  - IPC header and pseudo stack block (FP)
  - Buffers (temp and ring)
  - Shadow port table (SPTBL) (64.) of PIDs owning consoles

- PMGR has 6 tasks.
  - Five general purpose tasks. These tasks will run whatever portion of PMGR code necessary to complete current function, i.e. the tasks are not functionally restricted.
  - Timeout task. Initialization task becomes timeout task.

- Timeout task
  - If any timeouts active (TOCNT <> 0) timeout task will wake up every second (?DELAY) and scan chain of blocked PIB's (SUNB) for timeouts.
  - Can only timeout events that have block capabilities
  - Timeout task is highest priority
  - When no outstanding timeouts, timeout task is suspended and not doing ?DELAY's, therefore no timeout overhead unless timeouts are in progress.

- All devices have a control port and a read and/or write port.
  - Control port number obtained via ?ILKUP or ?GOPEN
  - Read/write port numbers returned each time device is opened.

- State Save Area (SSA)
  - Certain values in PIB 'pushed' to SSA upon passing of device from father to son.

- PMGR has own scheduler
  - Calls similar to RDOS calls (i.e. not ?TASK but .TASK)
  - PMGR has own bit to disable rescheduling in UST in order to disable rescheduling across system calls.
  - Frequent (over)use of INTDS/INTEN to avoid race conditions
  - Removes overhead of trapping into AOS for typical task functions
  - Supports a small subset of the available AOS tasking calls

- Cannot use ECLIPSE stack/linkage calls (like PSH, POP, SAVE)
  - Location 40 (SP) usually points to current PIB, and usually AC2 does too
  - Location 41 (FP) usually points to current IPC header/pseudo stack block (hence the name 'FP' for the block) and usually AC3 points to FP too
  - Locations 42 and 43 (CSL and CSO) are fair game and are not maintained across pends of calls
  - Locations in pseudo stack block used for return address storage, counters, temps, etc. And these locations are preserved across pends

9.8 PMGR - Card Reader Notes

- Hollerith to ASCII conversion is done on "text" reads
- Code set used is 029 code
- 026 code available by patching the conversion table.
- The card reader is "Started" for input when the device is opened
- The PMGR will "read ahead" as many cards as it can fit in the ring buffer.
- The card reader will not be started if there is not enough room for an entire card in the ring buffer. The assumption is made that a card needs 81 columns worth of room.
- The columns are not checked on input, so a mark-sense card reader would work also.
- Characteristics allow for binary reads to be "packed"
  - Four 12 bit columns packed in three 16 bit words.
- If characteristics not "packed"
  - 12 bits are right-justified in the buffer
  - Upper 4 bits used for status :
    - 100000= End of file
    - 040000= Hopper empty or stacker full
    - 020000= Pick fail
    - 010000= Read error.
- Characteristics allow for reads to be "trailing blanks suppressed"
  - PMGR inserts a NL character after the last non-blank character on that card.
  - Adjusts ring buffer memory, allowing another card to fit in sooner.
- A "parity error" on a text read from the card reader occurs when there was an illegal Hollerith code punched.

9.9 PMGR - Initialization

- Receives peripheral table (PERTB) from the AOS kernel as its initial IPC message
- Overwrites default PERTB which is provided in case no devices were sysgen'ed
- Turns superuser on so can write device entries into :PER
- Verifies self is at PID=1
- Sets up all PMGR tasks (i.e. Suspended and on USTAC chain)
- 2 pass initialization for data bases
  - Pass 1 computes lengths and starting addresses for data
  - Pass 2 actually builds data bases and does other init work
- Declares devices to system via ?DPMGR call
  - Causes DCT's to be set up
  - Causes DCU or IOP map to be set up
- Creates device entries into :PER
  - Different type of devices have different file types for their entries in :PER
- Builds line table from info in PERTB
- Builds PIBs
- Sets up special links for Pseudo Consoles, if any
- Initializes hardware if required (NIOC XX, etc)
- Initializes modem control signals
- Loads DCU or IOP if present
- Releases residual initialization memory

9.10 PMGR - DCU

- Runs with interrupts off
  - Software polling of interrupts
    1st - DCU local RTC
    2nd - MUX interrupts
    3rd - Host interrupts
    4th - DCU interrupts to host
  - Allows "Unpended" I/O within DCU monitor

- 2 level lock mechanism
  - 2 locations in PMGR space page zero (in host) used
    'HOLK' - Host lock word (0=unlocked)
    'DCLK' - DCU lock word (0=unlocked)
  - When DCU locked from host, host waits with interrupts disabled for DCU to acknowledge (Panics after 5 seconds)
  - DCU must finish all RTC and MUX interrupts before it can acknowledge the host (can be quite a while)

- DCU mapped to host using data channel MAP A
  - 1 slot to AOS page zero
  - 1 slot to PMGR page zero
  - N slots to PMGR data base start area through PMGR NMAX (approximately 15.)

- DCU initialized at PMGR init time
  - DCT set up on ?DPMGR call to AOS (processed in SOV15) which allocates MAP A slots at that time
  - All other data channel devices allocated MAP slots first and therefore there is a possibility of running out of MAP A slots
  - DCU local memory loaded by PMGR during PMGR init

- Host to DCU queue
  - Entries signify : (Queue of PMGR requests to DCU)
    - device start
    - modem off
    - modem on
  - Entries contain line number and device code
  - Queue contains one entry per line
  - Queue is in PMGR (host) address space

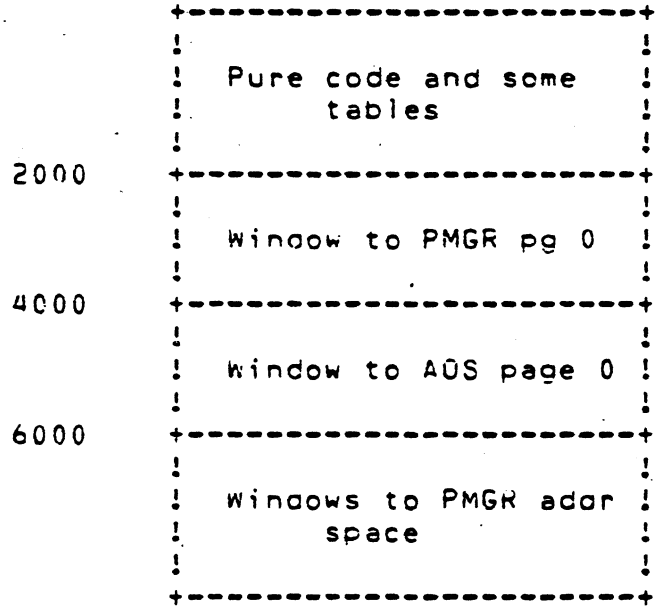
- DCU to Host queue
  - Entries for :
    - ^C^A, ^C^B, ^C^E
    - modem disconnect
  - Entries contain line # and device code
  - Queue is of fixed length
  - Queue is in DCU space

- Code in DCU is essentially the AOS modules PERDR.SR and MUXDR.SR without ECLIPSE instructions. (One character buffer in DCU)

- DCU handles character interrupts and inserts/removes characters from device ring buffers (through PIB pointers) in PMGR (Host) space DCU handles echoing.

- DCU normally executes code which resides in the DCU memory
  - DCU loader and MUX start routines used only during initialization reside in host memory (as does the powerfail restart code)

- DCU address space:



9.11 PMGR - IOP

- Entire PMGR is moved to IOP at IOPMGR init time
  - IOP DCT set up and IOP mapping (MAP D) set up during IOPMGR init's ?DPMGR call to AOS (processed in SOV15)
  - IOP memory filled with zeroes to ensure good parity, then each word from IOPMGR address space is deposited into the IOP and re-examined to verify good load

- IOP runs in mapped mode
  - One page mapped to AOS page zero
  - Two pages mapped to resident IOPMGR (called MART) in host
  - 29. pages mapped logical to physical in IOP
  - IOP MAP does not act like ECLIPSE MAP in that the MAP is not disabled upon an interrupt or SVC call
  - IOP memory parity is enabled
  - IOP MAP provides no defer protection (that is how regular PMGR panics)

- IOP/WART communication is through the MAP via the WARTS TCB's
  - TCB's in IOP logically locked 1 to 1 with TCB's in WART

- Resioual IOPMGR (WART)
  - Contains:
    - Tiny scheduler (just for SVC'ing into AOS)
    - 6 TCB's
    - 6 IPC headers with 255. word buffers for data
  - The WART executes System calls for the IOP that the IOP cannot execute itself (e.g. ?MBTU,?MBFU,?ISEND,?IREC,?SGNL)
  - WART's tiny scheduler gets control when IOP has loaded one of its TCB's and sets the TCB status to READY. The scheduler just SVC's into the system to issue call.
  - Host to IOP queue contains TCB address of completed System call and error/no error status bit. The IOP then extracts information from TCB and data from the associated buffer and completes the execution of the call for the task in IOP.
  - 2 K words in size

- IOP interrupts host for:
  - memory parity errors (fatal)
  - control sequences
  - modem status changes

- Host interrupts IOP for all entries posted onto "Host to IOP" Queue (i.e. Host TCB action completions)

- All IOPMGR data bases are in IOP
  - Certain words (e.g. Map words in line table, PIBs) are no longer needed because data bases are now in the address space of the interrupt drivers

- IOP has its own System call handler (SCALL)
  - Does as much of the system call processing as possible in the IOP (eg. The ?DELAY call is done entirely in the IOP)
  - System calls which can not be processed entirely in the IOP are broken into three parts:
    - Pre-processing in IOP
    - Transferring call to Host resident WART for execution
    - Post-processing in IOP (sometimes null)

- Most IOPMGR modules are common with regular PMGR modules

9.12 PMGR Databases

The four major PMGR data bases are :

Peripheral Table (PERTB)
Line Table (LTBL)
Peripheral Information Block (PIB)
IPC header and Pseudo stack block (FP)

9.12.1 'PERTB' - Peripheral Table

- Format is in IRMG2.SR found in :SYSGEN

- Obtained via ?IREC as initial IPC message during PMGR init

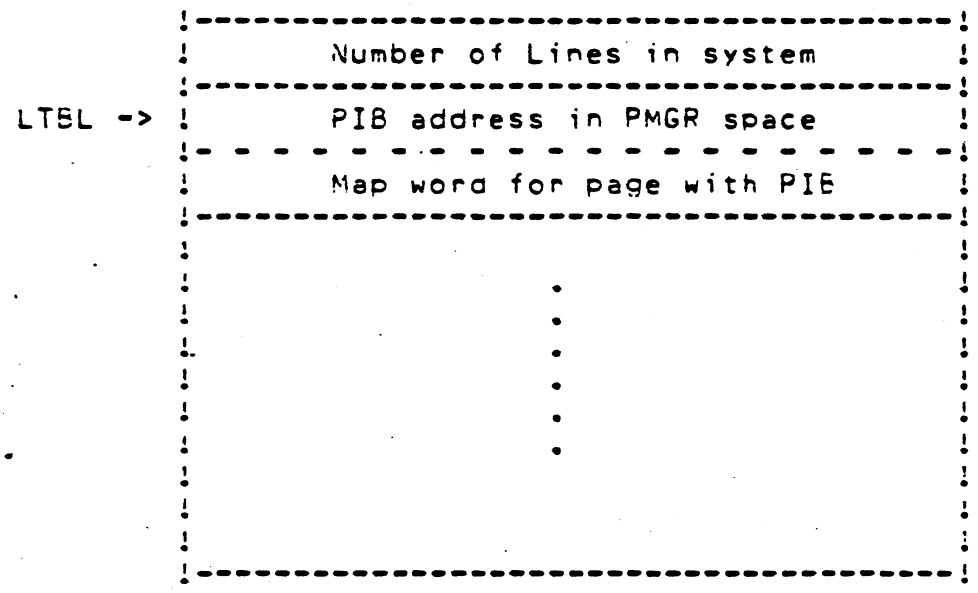
- Contains entries for all sysgen'ed PMGR supported devices
  - Device types
  - Device codes
  - Duffer sizes
  - Device characteristics

- Contains indication of presence of DCU or IOP

- Used only during PMGR init to build PMGR databases and then released along with residual PMGR address space

9.12.2 'LTBL' - Line Table

- Built at PMGR initialization time from PERTB information
- Word preceeding table is total number of ALM lines in table
- All entries two words long
  - PIB address for this line
  - Map word for page containing the PIB (for AOS interrupt world)
- Used by interrupt world to get to PIB



9.12.3 'PIB' - Peripheral Information Block

- Defined in PARS.LS
- One per device or one per line for multi-line devices
- Static, built during PMGR init and never released to memory pool
- Accessed by PMGR process, DCU(if present), and interrupt world
- Contains per device information like :
  - Device code(s)
  - Owner ID
  - Characteristics
  - Buffer information
  - Timeout information
- Points to ring buffer(S), temp buffer(s), and stack save area.

- PIB states are:
  - BLOCKED, e.g. Waiting for a buffer to empty or waiting for a modem connection
  - DONE, has become unblocked
  - ACTIVE, running

- Queue of PIBs in the BLOCKED state (waiting for some event like emptying of a buffer) or the DONE state (the buffer has emptied) is pointed to by 'SUNB'.

9.12.4 'FP' - IPC header and pseudo stack block

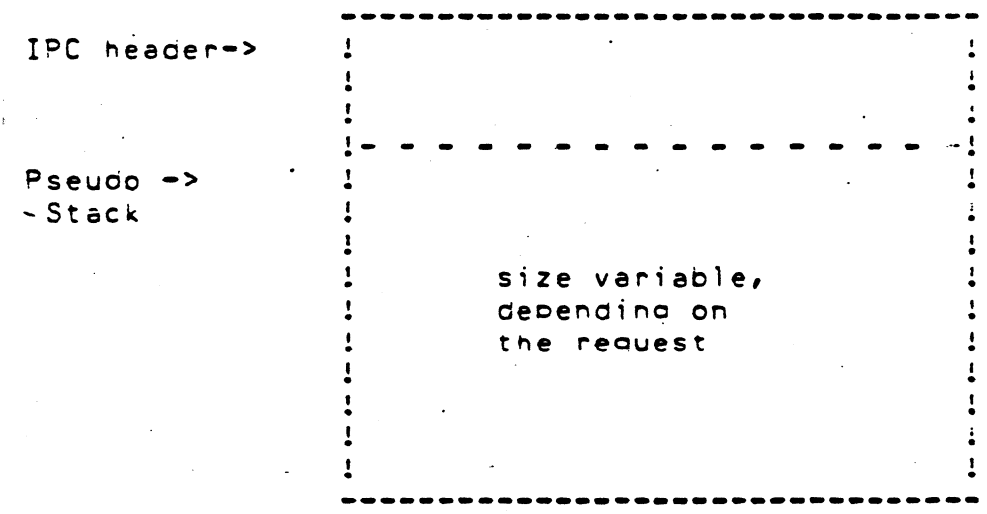
- Dynamically allocated

- One block allocated for every Control, Read or Write command

- Used as temporary storage in case path blocks

- Each offset is used by various routines for different reasons

- Block consists of standard IPC header followed by the pseudo stack proper



9.13 PMGR Control Functions

The following control functions are available :

value #	mnemonic	module name	function(s), result(s)
0*400	PMCOP	OPEN	Opens a device, returns I/O ports
1*400	PMCCL	CLOSE	Closes device, maybe deassigns
2*400	PMCAS	ASSIGN	Assigns a device
3*400	PMCRL	RELEAS	Deassigns a device
4*400	PMCAC	ASCON	Assign a console (system only)
5*400	PMCPT	PTERM	Terminate a process (system only)
6*400	PMCPC	CHAIN	Chain a process (system only)
7*400	PMCGT	GDTBL	Get a delimiter table
10*400	PMCST	SDTBL	Set delimiter table(s)
11*400	PMCGC	GDCHR	Get device characteristics
12*400	PMCSC	SDCHR	Set device characteristics
13*400	PMCSM	SEND	Send a message to a console
14*400	PMCRT	RTERM	Terminate I/O requests specified
15*400	PMCTO	SETIM	Set the time-out constant for a device
16*400	-----	-----	Reserved for future use
17*400	-----	-----	Reserved for future use

9.13.1 OPEN Control Function

Overview :

Increment open count. If first open, implicitly assign device, and assign memory buffers. Set modem DTR & RTS on, if not already connected. Output a FF or leader tape if characteristic specified. Read past leading nulls if characteristic specified.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCOP
?IDPH	from ?ILKUP or ?GOPEN
?IDPL	from ?ILKUP or ?GOPEN
?IOPN	a value from 1 to 177
?ILTH	0
?IPTR	(ignored)

Input to ?IREC :

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IDPN	(unchanged)
?ILTH	4
?IPTR	word address of a 4 word area in the PIB which has the read/write port numbers.

Output from ?IREC :

?ISFL	0			
?IUFL	[1b(RWERB)] + PMCOP			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	4	or	?ILTH	0
?IPTR	word address		?IPTR	error code

Notes : 1b(RWERB) = error bit, returned

9.13.2 CLOSE Control Function

Overview :

Decrement open count. If last close, kill all I/O, return error to all killed I/O requests. If implicit assign (and is last close), release device and buffers, turn modem off, release delimiter tables, restart output, reset default characteristics, reset default time-out value. Output trailer tape if characteristic specified.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCCL
?IDPH	from ?ILKUP or ?GOPEN
?IDPL	from ?ILKUP or ?GOPEN
?IDPN	a value from 1 to 177
?ILTH	0
?IPTR	(ignored)

Input to ?IREC : - unchanged

?ISFL	0
?IUFL	(unchanged)
?IDPH	(unchanged)
?IDPL	(unchanged)
?IDPN	(unchanged)
?ILTH	(unchanged)
?IPTR	(unchanged)

Output from ?IREC :

?ISFL	0			
?IUFL	[1b(RWERB)] + PMCCL			
?IDPH	(unchanged)			
?IDPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	0		?IPTR	error code

Notes : 1b(RWERB) = error bit, returned

9.13.3 ASSIGN Control Function

Overview :

Zero open count. If first assign, assign memory buffers, clear I/O requests, set default delimiter tables, assign device.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCAS
?IDPH	from ?ILKUP or ?GOPEN
?IDPL	from ?ILKUP or ?GOPEN
?IOPN	a value from 1 to 177
?ILTH	0
?IPTR	(ignored)

Input to ?IREC : - unchanged

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IDPN	(unchanged)
?ILTH	(unchanged)
?IPTR	(unchanged)

Output from ?IREC :

?ISFL	0			
?IUFL	[1b(RWERB)] + PMCAS			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	0		?IPTR	error code

Notes : 1b(RWERB) = error bit, returned

9.13.4 RELEASE Control Function

Overview :

Kill all I/O requests, return error to killed requests, release device and buffers. Zero open count, turn modem off, release delimiter tables, restart output, reset default characteristics, reset default time-out value.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCRL
?IDPH	from ?ILKUP or ?GOPEN
?IDPL	from ?ILKUP or ?GOPEN
?IUPN	a value from 1 to 177
?ILTH	0
?IPTR	(ignored)

Input to ?IREC : - unchanged

?ISFL	0
?IUFL	(unchanged)
?IUPH	(unchanged)
?IUPL	(unchanged)
?IDPN	(unchanged)
?ILTH	(unchanged)
?IPTR	(unchanged)

Output from ?IREC :

?ISFL	0			
?IUFL	[1b(RWERB)] + PMCRL			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	0		?IPTR	error code

Notes : 1b(RWERB) = error bit, returned

9.13.5 ASSIGN A CONSOLE Control Function (System only)

Overview :

Assign a device to son, set status as console. If first assign, assign memory buffers, set default delimiter tables, assign device, zero open count. If not first assign, kill all I/O, save current state (owner's PID, delimiter tables, time-out constant, open count), reset delimiter tables, and restart output.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCAC
?IDPH	0
?IDPL	0
?IOPN	0
?ILTH	0
?IPTR	Father's PID (left half), son's PID (right half)

Input to ?IREC : - none

Output from ?IREC : - none

Notes : Returns via ?SGNL

AC0=	0 or error code
AC1=	Son's PID
AC2=	0

9.13.6 TERMINATE A PROCESS Control Function (System only)

Overview :

Deassign all devices owned, except console. Close console. If father specified, reassign console to him, and restore previous state (owner's PID, delimiter tables, time-out constant, open count). Else clear console status, and deassign device.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCPT = 2400
?IDPH	0
?IDPL	0
?IOPN	0
?ILTH	0
?IPTR	Father's PID (left half), son's PID (right half)

Input to ?IREC : - none

Output from ?IREC : - none

Notes : returns via ?SGNL

AC0=	0
AC1=	Son's PID
AC2=	0

9.13.7 CHAIN A PROCESS Control Function (System only)

Overview :

Close all devices owned.

Input to ?ISEND :

```
?ISFL   0
?IUFL   PMCPC  = 3000
?IDPH   0
?IDPL   0
?IOPN   0
?ILTH   0
?IPTR   PID (right half)
```

Input to ?IREC : - none

Output from ?IREC : - none

Notes : Returns via ?SGNL

```
AC0=    0
AC1=    PID
AC2=    0
```

9.13.8 GET A DELIMITER TABLE Control Function

Overview :

Send a specified 16. word delimiter table to owner via IPC. User chooses input, output, or priority input delimiter table. Its 16. words are sent via IPC. Only one delimiter table may be obtained per call.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCGT [+1B(DTBLO)] [+1B(DTBLI)] [+1B(DTBLP)]
?IDPH	From ?ILKUP or ?GOPEN
?IDPL	From ?ILKUP or ?GOPEN
?IDPN	A value from 1 to 177
?ILTH	0
?IPTR	(ignored)

Input to ?IREC :

?ISFL	0
?IUFL	(unchanged)
?IDPH	(unchanged)
?IDPL	(unchanged)
?IDPN	(unchanged)
?ILTH	16.
?IPTR	Word address of 16. word buffer

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] +PMCGT [+1B(DTBLO)] [+1B(DTBLI)] [+1B(DTBLP)]			
?IDPH	(unchanged)			
?IDPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	16.	Or	?ILTH	0
?IPTR	Word address		?IPTR	error code

Notes :

1B(RWERB) = Error bit, returned
1B(DTBLO) = Output delimiter table
1B(DTBLI) = Input delimiter table
1B(DTBLP) = Priority input delimiter table

Format of delimiter table is described in AOS programmer's manual, chapter 6.

9.13.9 SET DELIMITER TABLES Control Function

Overview :

Set a 16. word delimiter table from owner via ?MBFU. User chooses input and/or output and/or priority input delimiter table(s) to be set. Any or all delimiter tables may be set in one call.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCST [+1B(DTBLO)] [+1B(DTBLI)] [+1B(DTBLP)]
?IDPH	From ?ILKUP or ?GOPEN
?IDPL	From ?ILKUP or ?GOPEN
?IOPN	A value from 1 to 177
?ILTH	0
?IPTR	Word address of 16. word buffer containing delimiter table

Input to ?IREC :

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IDPN	(unchanged)
?ILTH	0
?IPTR	(ignored)

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] +PMCST [+1B(DTBLO)] [+1B(DTBLI)] [+1B(DTBLP)]			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	0		?IPTR	error code

Notes :

1B(RWERB) = Error bit, returned
1B(DTBLO) = Output delimiter table
1B(DTBLI) = Input delimiter table
1B(DTBLP) = Priority input delimiter table

Format of delimiter table is described in AOS programmer's manual, chapter 6.

9.13.10 GET DEVICE CHARACTERISTICS Control Function

Overview :

Send 3 words of device characteristics to owner via IPC. User receives 3 words of current or default device characteristics. Default characteristics may be obtained by anyone if the device is not open, else only by PID 2 if the device is open. Current characteristics may by obtained only by the owner.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCGC [+1B(CHBDF)]
?IDPH	From ?ILKUP or ?GOPEN
?IDPL	From ?ILKUP or ?GOPEN
?IOPN	A value from 1 to 177
?ILTH	0
?IPTR	(ignored)

Input to ?IREC :

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IOPN	(unchanged)
?ILTH	3
?IPTR	Word address of 3 word buffer to receive characteristics

Output from ?IREC :

?ISFL	0		
?IUFL	[1B(RWERB)] + PMCGC [+1B(CHBDF)]		
?IOPH	(unchanged)		
?IOPL	(unchanged)		
?IOPN	(unchanged)		
?ILTH	3	or	?ILTH 0
?IPTR	Word address		?IPTR error code

Notes : 1B(RWERB) = Error bit, returned
1B(CHBDF) = Default characteristics
0B(CHBDF) = Current characteristics

Format of device characteristics described in "PARU.SR" and in the AOS Programmer's Manual, chapter 6. Default characteristics are from sysgen ("AOSGEN.PR"). Current characteristics are from last characteristics change.

9.13.11 SET DEVICE CHARACTERISTICS Control Function

Overview :

Get 3 words of device characteristics from owner via IPC. User sends 3 words of default or current characteristics via IPC. Default characteristics may be set only by PID 2. Current characteristics may be set only by owner. Start/stop timeouts if enabled/disabled.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCSC [+1B(CHBDF)]
?IDPH	From ?ILKUP or ?GOPEN
?IDPL	From ?ILKUP or ?GOPEN
?IOPN	A value from 1 to 177
?ILTH	3
?IPTR	Word address of 3 word buffer with new characteristics

Input to ?IREC :

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IDPN	(unchanged)
?ILTH	0
?IPTR	(ignored)

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] + PMCSC			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	0		?IPTR	error code

Notes : 1B(RWERB) = Error bit, returned
1B(CHBDF) = Default characteristics
0B(CHBDF) = Current characteristics

Format of device characteristics described in "PARU.SR", and the AOS Programmer's Manual, chapter 6. Default characteristics are normally from sysgen ("AOSGEN.PR"). Current characteristics are retained until changed or the device is released.

9.13.12 SEND Control Function

Overview :

Send a maximum of 255. text bytes to a console device. PMGR precedes the text with "FROM PID ### : ". The message resets ^O if it is set on the console. The request is put on the specified device's write Queue, so it could pend for a long time or be terminated by a close.

Input to ?ISEND :

```
?ISFL   0
?IUFL   PMCSM + # bytes to send (255. max) [+1B(RWIOG)]
?IDPH   From ?ILKUP or ?GOPEN
?IDPL   From ?ILKUP or ?GOPEN
?IOPN   A value from 1 to 177
?ILTH   0
?IPTR   Byte address of text buffer to send
```

Input to ?IREC : - unchanged

```
?ISFL   0
?IUFL   (unchanged)
?IOPH   (unchanged)
?IOPL   (unchanged)
?IDPN   (unchanged)
?ILTH   (unchanged)
?IPTR   (unchanged)
```

Output from ?IREC :

```
?ISFL   0
?IUFL   [1B(RWERB)] + # bytes sent [+1B(RWIOG)]...
        ...[+1B(RWFTW)] [+1B(RWTTO)] + 15(RWPRF)
?IOPH   (unchanged)
?IOPL   (unchanged)
?IDPN   (unchanged)
?ILTH   0               or      ?ILTH   0
?IPTR   0                       ?IPTR   0
```

Notes : 1B(RWERB) = Error bit, returned

- See "write options" described later in this document.

- The entire message is "<012>from PID ### : (message)". The User text should end with a NL (012).

9.13.13 REQUEST TERMINATION Control Function

Overview :

I/O requests from the specified User ports are aborted with no error returned to them. PMGR returns to the caller the number of requests actually terminated.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCRT
?IDPH	From ?ILKUP or ?GOPEN
?IDPL	From ?ILKUP or ?GOPEN
?IOPN	A value from 1 to 177
?ILTH	2
?IPTR	Word address of global ports of I/O requests to terminate

Input to ?IREC :

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IOPN	(unchanged)
?ILTH	0
?IPTR	(ignored)

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] + PMCRT			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IOPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	# aborted		?IPTR	error code

Notes : 1B(RWERB) = error bit, returned

- Format of global ports of I/O requests to terminate :

Word address	+0 :	0/PID
	+1 :	?IOPN

- Use ?IOPN of User's read or write port needing termination. If there is an ?IREC on those ports, it will not be answered with an abort message. Abort that ?IREC also.

9.13.14 SET TIME OUT CONSTANT Control Function

Overview :

Set a new value (or default value if -1) for the time out constant. PMGR returns the actual value set in all cases. Values of 0 or 1 are not allowed and are set to 2. Values are in integer seconds.

Input to ?ISEND :

?ISFL	0
?IUFL	PMCTO
?IDPH	From ?ILKUP or ?GOPEN
?IDPL	From ?ILKUP or ?GOPEN
?IOPN	A value from 1 to 177
?ILTH	0
?IPTR	Value, in seconds

Input to ?IREC : - unchanged

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IDPN	(unchanged)
?ILTH	(unchanged)
?IPTR	(unchanged)

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] + PMCTO			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	New value		?IPTR	error code

Notes : 1B(RWERB) = error bit, returned

- Time outs are enabled/disabled by setting/clearing the time out bit in the device characteristics words.

9.13.15 GET STATISTICS Control Function

Overview :

Returns via IPC 9. words of PMGR internal runtime statistics. This feature is not documented, and not supported.

Input to ?ISEND :

?ISFL	0
?IUFL	16*400
?IDPH	From ?ILKUP or ?GOPEN of @CON0
?IDPL	From ?ILKUP or ?GOPEN of @CON0
?IOPN	A value from 1 to 177
?ILTH	0
?IPTR	(ignored)

Input to ?IREC :

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IOPN	(unchanged)
?ILTH	9.
?IPTR	Word address of 9. word buffer to receive statistics

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] + 16*400			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IOPN	(unchanged)			
?ILTH	9.	Or	?ILTH	0
?IPTR	Word address		?IPTR	error code

Notes : 1B(RWERB) = error bit, returned

- Format of PMGR internal runtime statistics (in PMGR page zero)

Word	Mnemonic	Loc.	Definition
0	RDHI	26	Number of 2**16 bytes input
1	RDLS	27	Number of bytes input
2	WRHI	30	Number of 2**16 bytes output,including read echoes
3	WRLO	31	Number of bytes output, including read echoes
4	CLHI	32	Number of 2**16 control requests (open, etc.)
5	CLLO	33	Number of control requests, even not completed
6	SCHI	34	Number of 2**16 system calls, incl. reschedules
7	SCLO	35	Number system calls, including rescheduling (lo)
10	OPCT	36	Number of devices now open

9.13.16 RESET STATISTICS Control Function

Overview :

Zeroes : RDHI, RDLO, WRHI, WRLO, CLHI, CLLO, SCHI, SCLO. Does not zero OPCT. This feature is not documented and not supported.

Input to ?ISEND :

?ISFL	0
?IUFL	17*400
?IDPH	From ?ILKUP or ?GOPEN of @CON0
?IDPL	From ?ILKUP or ?GOPEN of @CON0
?IOPN	A value from 1 to 177
?ILTH	0
?IPTR	(ignored)

Input to ?IREC : - unchanged

?ISFL	0
?IUFL	(unchanged)
?IDPH	(unchanged)
?IDPL	(unchanged)
?IDPN	(unchanged)
?ILTH	(unchanged)
?IPTR	(unchanged)

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] + 17*400			
?IDPH	(unchanged)			
?IDPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	0		?IPTR	error code

Notes : 1B(RWERB) = error bit, returned

- OPCT is NOT reset to zero.
- For description of statistics, see "GET STATISTICS" on the previous page of this document.

9.14 PMGR I/O Functions

The following I/O functions are available :

Label	Module	Function
READ	IOCOM	Read byte(s) per function bits and characteristics
WRITE	IOCOM	Write byte(s) per function bits and characteristics
SCIN	SCEDT	Screenedit read byte(s)

9.14.1 READ I/O Function

Overview :

Reads data into a specified User buffer. Data is transferred via ?MBTU. Absolute maximum number of bytes is 511. in one request.

Input to ?ISEND :

?ISFL	0
?IUFL	[read options] + [max # bytes (777 mask) or 0]
?IDPH	From open
?IDPL	From open
?IOPN	A value from 1 to 177
?ILTH	0
?IPTR	Byte address of buffer to receive data

Input to ?IREC : - unchanged

?ISFL	0
?IUFL	(unchanged)
?IDPH	(unchanged)
?IDPL	(unchanged)
?IDPN	(unchanged)
?ILTH	(unchanged)
?IPTR	(unchanged)

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] + [read options] + actual # bytes input			
?IDPH	(unchanged)			
?IDPL	(unchanged)			
?IDPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	0		?IPTR	error code

Notes : 1B(RWERB) = error bit, returned

- Number of bytes input includes delimiter, but not EOF (^D).

- Read options

  ?IUFL read options on input are :

Flag Name	Bit Position	Flag Description When Bit = 1
RWIOG	0.	I/O Buffer is in GHOST context
RWERB	1.	Don't echo single character delimiters
RWDRP	2.	Drop typed-ahead characters before read
RWRNE	3.	Read with no echo
RWBTF	4.	Binary transfer flag
RWTTQ	5.	Transfer to byte quantity specified (non data-sens.)
RWPRF	6.	Priority read flag
	7.-15.	Maximum number of bytes to read

- Read conventions are :
  - ?IUFL field for number of bytes :
    If max # bytes specified in ?IUFL is 0, and RWTTQ = 0 (read to delimiter), then max # bytes is defaulted to ?DFLL.
  - ?DFLL = 210, =136. Defined in "PARU.SR".

- Priority reads :
  - Priority reads will preempt any non-priority reads after the current character of input.
  - No non-priority read will get input as long as there are any priority reads outstanding.
  - Multiple priority reads are enqueued FIFO before any multiple non-priority reads, which are also enqueued FIFO.

- Binary mode input :
  - The first binary read puts the device in "binary mode".
  - In binary mode, control characters, control sequences, and read controls are non-functional and passed directly to the USER.

- Binary mode input does not :
  - Echo
  - Interpret single control characters: ^P, ^Q, ^S, etc
  - Interpret double control characters: ^C^A, ^C^B, etc
  - Interpret ^U
  - Interpret ^D
  - Interpret rubout
  - Interpret interrupt level characteristics: NAS, etc
  - Interpret read level characteristics: UCO, ULC, etc

- Binary input mode is exited by :
  - Hitting the "BREAK" key on a Console terminal
  - The next non-binary read
  - Process termination

9.14.2 WRITE I/O Function

Overview :

Writes data from a specified User buffer. Data is transferred via ?MBFU. Absolute maximum number of bytes is 511. in one request.

Input to ?ISEND :

?ISFL	0
?IUFL	[write options] + [max # bytes (777 mask) or 0]
?IDPH	From open
?IDPL	From open
?IOPN	A value from 1 to 177
?ILTH	0
?IPTR	Byte address of buffer containing data to write

Input to ?IREC : - unchanged

?ISFL	0
?IUFL	(unchanged)
?IOPH	(unchanged)
?IOPL	(unchanged)
?IDPN	(unchanged)
?ILTH	(unchanged)
?IPTR	(unchanged)

Output from ?IREC :

?ISFL	0			
?IUFL	[1B(RWERB)] + [write options] + actual # bytes output			
?IOPH	(unchanged)			
?IOPL	(unchanged)			
?IOPN	(unchanged)			
?ILTH	0	or	?ILTH	0
?IPTR	0		?IPTR	error code

Notes : 1B(RWERB) = error bit, returned

- Number of bytes output includes the delimiter.

- Write options are :

Flag Name	Bit Position	Flag Description When Bit = 1
RWIUG	0.	I/O Buffer is in GHOST context
	1.	(Reserved)
	2.	(Reserved)
RWRNE	3.	Do a forced write (clear ^O)
RWBTF	4.	Binary transfer flag
RWTTQ	5.	Transfer to byte quantity specified (non data-sens.)
	6.	(Reserved)
	7.-15.	Maximum number of bytes to read

- Write conventions are :
  - Binary/Text OUTPUT has no effect on "binary INPUT mode", described under "Read conventions" earlier in this document.
  - Binary OUTPUT does not use any device characteristics.

- ?IUFL number of bytes field :
  - If output is "to delimiter", and the max # bytes is 0, then a maximum of ?DFLL is defaulted.
  - ?DFLL =210, =136. Defined in "PARU.SR".

9.14.3 SCREEN EDIT I/O Function

Overview :

Optionally displays the buffer, optionally positions the cursor in the displayed string, then does a "Screen-edit read" into the same User buffer. Data is transferred via ?MBFU and ?MBTU. Absolute maximum number of bytes is 511. in one request.

Input to ?ISEND :

?ISFL	0
?IUFL	[read options] + [max # bytes (777 mask) or 0]
?IDPH	From open
?IDPL	From open
?IUPN	A value from 1 to 177
?ILTH	3
?IPTR	Word address of 3 word packet

Input to ?IREC :

```
        ?ISFL   0
        ?IUFL   (unchanged)
        ?IOPH   (unchanged)
        ?IOPL   (unchanged)
        ?IDPN   (unchanged)
        ?ILTH   3
        ?IPTR   Word address of a 3 word packet
```

Output from ?IREC :

```
        ?ISFL   0
        ?IUFL   [1B[RWERB)] + [read options] + actual # bytes input
        ?IOPH   (unchanged)
        ?IOPL   (unchanged)
        ?IDPN   (unchanged)
        ?ILTH   3
        ?IPTR   Word address of a 3 word packet
```

Notes : 1B(RWERB) = error bit, returned

- Number of bytes input includes delimiter, but not EOF (^D).

- Format of 3 word packet on ISEND
  packet 0 : col/row
  1 : flags and edit position
  2 : bytepointer to buffer

- Format of 3 word packet on IREC
  packet 0 : error code (if error flag set in ?IUFL)
  1 : absolute cursor position (col/row) on I/O done
  2 : reserved

- Format of "Initial cursor position in (displayed) string" :
  -1 = No display requested
  0 = Position cursor at beginning of string after display
  "N" = Position cursor after "N"th char in displayed string

- Line edit ?IUFL read options :

Flag Name	Bit Position	Flag Description When Bit = 1
RWIOG	0.	I/O Buffer is in GHOST context
	1.	(Reserved)
	2.	(Reserved)
	3.	(Reserved)
	4.	(Reserved)
	5.	(Reserved)
RWPRF	6.	Priority read flag
	7.-15.	Maximum number of bytes to read

- Screen-edit read conventions :
  - Screen-edit reads are "echo", "text", and "to delimiter".
  - See notes on "max # bytes", and "read to delimiter" under "read options" on a previous page of this document.

9.15 PMGR - Control Characters

The following is a listing of control characters used by PMGR :

- Single character, interrupt level, controls :

^C	<003>	Start double character control sequences. - The next character determines action taken.
^O	<017>	Drop all future output until : - The next read - The next "forced output" write - ^Q - Process termination
^P	<020>	The next single character after ^P is not interpreted as a single character control.
^Q	<021>	Continues output suspended by ^S or dropped by ^O
^R	<022>	(not reserved, is available to program)
^S	<023>	Stops all future output
^T	<024>	(reserved, is a no-op now)
^V	<026>	(reserved, is a no-op now)

- Single character, read level, controls :

^D	<004>	End-of-file. Not passed to program.
^U	<025>	Erase input. Not passed to program.

- Double character, interrupt level, control sequences :

^C^A	<003><001>	Interrupt
^C^B	<003><002>	Process terminate
^C^C	<003><003>	(reserved, is a no-op now)
^C^D	<003><004>	(reserved, is a no-op now)
^C^E	<003><005>	Process terminate and create a break file.

THIS PAGE ORIGINALLY WAS LEFT BLANK

CHAPTER 10 - EXEC
(updated for AOS rev. 3.11)

10.1 The EXEC Tasks

10.1.1 Overview

After initialization EXEC has nine tasks, each with a unique responsibility. Although most EXEC subroutines are used by several of the tasks, each task has a distinct "main loop". The default task creates the other eight tasks at EXEC initialization time and they remain in existence until the EXEC is terminated.

The logon device handler task (which develops from the initial task) does all of the EXEC's console reads and writes during the logon dialogue and creates processes for console users.

The terminate detector task listens for process termination messages from the system and notifies either the console driver task or the batch dequeuer task of the terminations. The terminate detector also displays coop termination messages on the operator console.

The dequeuer task looks for queue entries that need to be processed and for idle batch streams and coops to process the entries.

The mount manager task monitors the mount descriptor chain looking for entries which need servicing by the operator, displaying appropriate instructions to the operator and waiting for the command decoder task to supply the operator's response.

The command decoder task processes the "CONTROL @EXEC" commands.

The request decoder task processes the ?EXEC system call, usually making or changing queue entries or manipulating the mount descriptor list. The request decoder notifies the dequeuer or the mount manager whenever it has done something of interest to them.

The coop listener task listens for messages from the coops. These messages are of two types, requests to display messages on the operator console, and requests for another file to process.

The IPC ignorer task is responsible for receiving and discarding messages to certain of the EXEC's IPC ports to prevent the clogging of EXEC's IPC spool file. The EXEC creates an IPC entry of type ?FQUE for each of the queues that is open so that the GHOST knows what queues are available, but no messages should ever be sent to the ports corresponding to these entries (as a matter of fact, all ?FQUE entries created by the EXEC have same port number).

The delay manager task monitors the delay descriptor chain and processes entries when they expire.

10.1.2 The Logon Device handler task

Module name: EXLOG.SR
Entry point: LDGO

Console Handling

The EXEC was designed to handle an "unlimited" number of consoles. This capability could not be achieved by issuing ?OPEN/?READ/?WRITE and ?CLOSE level system calls since these calls all block the issuing task until the requested operation is complete thus requiring a task to be dedicated to each console. Instead, the EXEC uses IPC messages directly with the peripheral manager which allows an I/O operation to be initiated and the issuing task to proceed immediately to some other operation.

Centralized Control

Control of all consoles is centralized in order to easily deal with multiple asynchronous console related events. The primary events of this type are modem disconnects, I/O timeouts, and disable commands from the operator. This facility is accomplished by allowing one and only one task to deal with consoles.

Memory Utilization

The memory requirements of handling each console are minimized so that a large number of consoles can be handled. Each console requires a logon descriptor (LD) and a stack. This capability is achieved by handling each console in a manner which requires only temporary stack usage. Whenever any I/O is being done on a particular console a stack is required. However, the only instance where this could be a long period of time is during the actual logon dialogue sequence. Consequently, no stack requirement exists either while the console is in the "**** type newline to begin logging on ****" state or while the console is actually logged on.

Operation

Whenever a console either initiates an I/O operation on a console or creates a process for a console, the console driver task calls a special routine "WTLD". This routine saves the currently active stack control locations (40-43) for the console being processed and switches to a default stack which is not assigned to a particular console. A check is then made to determine whether the stack in use for the console being processed can be released. The stack can be released if it contains only a return block and no stack temps. It is important that this condition always exist except during the logon dialogue which has a finite, brief, duration. When the stack is releasable, the return block is copied to

the LD and the stack itself is released to free memory. If the stack is not releasible, the stack control locations for a specific console are copied to the LD and the stack is not released.

Since the EXEC issues all console IPC messages from the same local port, WTLD then issues an ?IREC for any message to that port and suspends. When a message is received, the linked list of LD's (LDCHN) is searched to find the console associated with the particular message.

When the appropriate LD is found, a check is made to see whether a stack already exists. If so, the stack control locations are copied from the LD to locations 40-43 and a return is done. If no stack is allocated, stack space is obtained from free memory, the return block is copied to it form the LD, locations 40-43 are set appropriately, and a return is made to the caller.

Processing then proceeds until an I/O operation is initiated and WTLD is called again.

Initiation of I/O operations is achieved through the I/O subroutines by setting up an IPC header in the LD with message transfer postponed until WTLD is called. This is done so that in certain cases the ?ISEND can be postponed until some additional event transpires. This occurs primarily when a process is created for a console since process termination has to be detected by a different task (the terminate detector). The process creation routine (CPROC in EXSUB.SR) creates the process and sets a special bit (BTNIS) in the LD to inhibit the ?ISEND the next time WTLD is called. CPROC then returns and the console driver task initiates the I/O operation that causes a newline to be output at the beginning of a logoff message and WTLD is called. WTLD observes that BTNIS is sent and omits the ?ISEND portion of its operation. Note that at this point the LD has an IPC header already to go to write that newline and that the terminate detector task need merely issue an ?ISEND when the process terminates.

10.1.3 The Terminate Detector Task

Module Name: EXEC.SR
Entry Point: LGOF

The terminate detector task suspends on an ?IREC from global port ?SPTM from which eminate termination messages for all processes. The IPC message received describes the details of the process termination and reason for termination. Taking the PID from the message, the terminate detector searches through the logon device chain (LDCHN), the batch stream chain (BSCHN) and the coop descriptor chain (CDCHN) looking for a matching PID. Depending on which chain the PID occurs, appropriate actions are initiated.

If the process was a coop, the terminate detector ?SENDs a message informing the operator that the particular coop has terminated.

If the process was a console job, an ?ISEND to the PMGR is issued on the IPC header already set up in the LD . The terminate detector then waits on an ?XMT/?REC lock for the console driver task to finish using the contents of the process termination message buffer to display the process termination message on the console affected. The console driver task then undoes the ?XMT/?REC lock.

If the process was a batch job, its output file is reopened, the process termination message is written to it, the list file is QPRINTed (if non-empty), the output file is closed and QPRINTed, and the dequeuer task is "poked".

The terminate detector then loops back to its ?IREC.

10.1.4 The Dequeuer Task

Module Name: XDEQ.SR
Entry Point: DEQ

The dequeuer task suspends on an ?XMT/?REC mailbox waiting to be "awakened". No useful data is actually passed through the mailbox, it is a convenient way to pend this task when there is nothing for it to do. Whenever another task puts an entry into the queues or modifies a queue entry, or whenever a queue server (coop or batch stream) goes idle, the dequeuer task is "awakened" by doing an ?XMT to its mailbox. Since more than one task can do this and since the goal is to unpend the dequeuer task, these awakening tasks ignore the "attempt to ?XMT to mailbox in use" error.

The dequeuer task searches the batch streams (BSCHN) and the coops (CDCHN) looking for a slot to place the queue entry. For each batch stream or coop that it finds, the dequeuer task attempts to select from the queues (.QSEL) the best possible entry for processing.

The dequeuer then loops back to its mailbox.

10.1.5 The Mount Manager Task

Module Name: XREQ.SR
Entry Point: MNT

The mount manager task is complicated by the fact that in certain case it has to synchronize with the command decoder task when it needs a response from the operator.

The mount manager task idles waiting to be continued by the request decoder task. When such an even occurs, this signifies that the request decoder task has either created or modified a mount descriptor (MD).

The mount manager task then searches the MD chain (MDCHN) for entries which need processing. For each entry, it ?SENDs instructions to the operator's console, and delays. When the delay expires, the mount manager repeats this sequence. If the command decoder task receives a "mounted" or "refused" command from the operator it verifies that a mount in progress flag is set, and if so extracts information from the "mounted" or "refused" command and issues an ?IDGUTD to cancel the mount manager's ?DELAY request. It then waits on an ?XMT/?REC mailbox for the mount manager to handle the command.

The mount manager then updates the MD appropriately, unpends the command decoder task, and using information stored in the MD by the request decoder task returns status to and unpends the user program which originally issued the ?EXEC call.

The mount manager then looks for another MD to handle until it exhausts MDCHN and then waits to be reawakened.

10.1.6 The Command Decoder Task

Module Name: EXCON.SR
Entry Point: CON

The command decoder task decodes "CONTROL @EXEC" commands and calls the appropriate command handlers (located in XCOV0-XCOV9).

The command decoder task waits on an ?IREC from any global port to the local port corresponding to the IPC entry @EXEC created by the EXEC at initialization. When a message is received, the process ID and username of the sender is obtained. As part of its initialization, the EXEC obtained its own user name. When these two user names match, the sender is considered to be "the operator" and the command will be decoded. Otherwise the command decoder task ?SENDs an error message to the sender.

When the sender is the operator, the command is found in the command table. Byte pointers to the arguments together with the number of arguments are then pushed onto the stack, and the appropriate command processing routine is called. Each of these routines begins by verifying that a legal number of arguments was supplied. This check occurs in each command routine because each of these routines is an overlay. Thus, the argument requirements for each command exist in the source module containing it rather than in a table in the root. Since ?RCALLs are used, the overlay cannot be preloaded for argument checking by the root code.

When a command routine is dispatched, AC0 is cleared. The command invokes the command decoder to output a message to the issuer by returning a system error code in AC0. The message is not interpreted as an error message. Command processors also can ?SEND messages directly to the issuer of the command.

At the completion of the command, if prompts are "on" the time of day is ?SEND to the issuer and the command decoder task awaits another IPC message.

The command decoder stores the process ID of the issuer in ?USP (location 16). ?USP is context switched by the system each time it switches tasks making itself unique for each task. The ?SEND logic in the EXEC directs messages to the process whose ID that it finds in ?USP. Thus, the responses from the commands are returned to the console from which the "CONTROL @EXEC" was issued. However, only the direct responses from a command will be ?SEND to the issuer. For example, if a batch stream were continued, the "continuing" message would be returned. However the "starting" message and the like would be sent to the console from which the EXEC was ?PROC'd.

10.1.7 The Request Decoder Task

Module Name: XREQ.SR
Entry Point: REQ

The request decoder task is responsible for primary processing of the ?EXEC system call. The following describes the implementation of this system call.

?EXEC is a GHOST call which performs an IPC dialogue with the EXEC. There are two important points to consider. First, a user without the (dangerous) IPC privilege can call the EXEC in a controlled manner. Secondly, the user can pend waiting for the EXEC to finish the call and unpend the caller.

The ?EXEC handler in the GHOST does an ?ILKUP of @EXEC_REQUEST, an entry of type ?FIPC created by the EXEC during its initialization. Every GHOST must perform a ?ILKUP in handling a ?EXEC call. This is

required because the EXEC may have gone down and come back up with a different process ID. This may cause a different global port number for @EXEC_REQUEST to be assigned (this applies only for those GHOSTS not under the EXEC). If the GHOST's ?ILKUP of @EXEC_REQUEST fails, the GHOST automatically gives an error return to the user's ?EXEC with error code for "EXEC NOT AVAILABLE".

When the GHOST does find @EXEC_REQUEST, it builds an IPC header with the contents of the first word of the user's ?EXEC packet in offset ?IUFL, the global port from the ?ILKUP in offsets ?IDPH and ?IDPL, the address of the user's ?EXEC packet in offset ?IPTR, and a 0 in offset ?ILTH. However, the packet itself is not sent, since the GHOST cannot determine the packet size (in fact, the length varies for each ?EXEC function). The first word of the ?EXEC packet is sent since it is the function and the request decoder task can figure out from that how big the packet is. The address of the packet is sent so that the request decoder task can fetch a copy of the packet via ?MBFU (move bytes from user).

Using the header it has constructed, the GHOST does a ?IS.R system call to send the message and wait for a response. ?IS.R is used rather than ?ISEND/?IREC because the request decoder task always does its ?ISEND with the "do not spool if no receiver ready" system flag set while ?IS.R guarantees that a receiver is ready. The request decoder task sets this flag in case the ?EXEC issuer has done a ^C^A and is no longer waiting. Otherwise, the message were to be spooled, the user's next ?EXEC call would get the spooled response from the aborted task. Subsequent messages would be spooled causing synchronization problems.

When the GHOST returns from its ?IS.R, it checks to see if offset ?IPTR is non-zero. If so, the value therein is an error code, the value is passed to the user's AC0. The user is unpended at the exception return from the ?EXEC call. If the value is zero, the user's AC's are unchanged and the user is unpended at the normal return from the ?EXEC call.

The request decoder task idles on an ?IREC from any global port to the local port corresponding to the IPC entry @EXEC_REQUEST which the EXEC creates during its initialization. When a message is received, offset ?IUFL is checked to see that it contains a valid ?EXEC function code. If not, the error "UNKNOWN ?EXEC FUNCTION" is returned as described above. If so, the length of the packet for the particular function is searched in a table and the user's packet is copied into the EXEC via the ?MBFU (move bytes from user) system call. The request decoder task then dispatches to the appropriate request processor via an ?RCALL (these processors are contained in overlays). When the processor is called, AC0 is set to zero. The processors indicate errors by returning with AC0 non-zero; the non-zero value is a system error code which is passed back to the user as described above.

There are two ways that request functions unpend the user. First, there is the normal request which can be serviced immediately and therefore the request decoder task can issue the ?ISEND back to the GHOST immediately upon return from those request processors. Secondly, the less usual case is for request functions which require operator intervention (specifically, tape mount/dismount requests). In these cases, the request processors set the no reply flag (NOREPL) and the request decoder task omits its ?ISEND. When the operator has finished handling the request, the mount manager task will issue the ?ISEND back to the user. This is done to prevent the holding up of GPRINTs and the like while waiting for the operator to mount a tape.

10.1.8 The Coop Listener Task

Module Name: EXEC.SR
Entry Point: CTX

The coop listener task idles on an ?IREC from any global port to a local port. The global version of the particular local port is passed to each coop at the time coop is ?PROC'd. Each message is verified that it came from a coop process, otherwise the message is discarded. Messages are of three types and are sorted out as follows:

type 1 - request to display a system error message. In this case, offset ?IUFL contains a zero, ?IPTR contains an error code, and ?ILTH contains a zero.

Type 2 - request to display a text string. In this case, offset ?IUFL also contains a zero, but the IPC message body is present and contains a null-terminated text string to be displayed.

Type 3 - notification that the coop has finished processing its current file. In this case, offset ?IUFL contains a minus one, and ?IPTR and ?ILTH both contain zero.

10.1.9 The IPC Ignorer Task

Module Name: EXEC.SR
Entry Point: IGN

The IPC ignorer task ignores IPCs. The EXEC creates an IPC entry of type ?FQUE for each of the queues that is open (such as @LPT). These entries are created to indicate which queues are available. IPC messages will never be sent to the ports corresponding to these entries (in fact they all are given same local port). This task idles on an ?IREC from any global port to the local port. Each time the ?IREC unsuspends, it is immediately re-issued. This mechanism discards all messages and prevents a user from filling the IPC spool file for the EXEC.

10.1.10 The Delay Manager Task

Module Name: XDELY.SR
Entry Point: DTSK

The delay manager task manages requests for processing at a specific time and date or after a specified amount of time has elapsed. These requests are primarily generated for the /AFTER queuing feature, and for delaying console I/O such as "TOO MANY ATTEMPTS, CONSOLE LOCKING FOR 10 SECONDS".

Other EXEC tasks create delayed processing requests by calling either of two subroutines, "DELAY" and "UNTIL" in XDELY.SR. When the delay expires, these routines require both the address of a routine for the delay manager task to "JSR to" and a datum to be passed as input to that routine. The difference between the two routines is in the manner of specification of the delay's duration . "DELAY" accepts a number of seconds to delay while "UNTIL" accepts a date and time (in file system format) at which to continue processing.

The DELAY and UNTIL routines create delay descriptors (DD) and insert them in sorted order (first expiration first) on a linked list (DDCHN) which is protected by an ?XMT/?REC lock (DDLOK) and then issue an ?IDGOTO to unpend the delay manager task from ?DELAY which it idles on.

Upon the expiration of its ?DELAY or being ?IDGOTO'd, the delay manager task acquires DDLOK, reads the current time and date, and searches DDCHN for delay descriptors whose time has expired. For each DD which needs processing, the delay manager task calls the specified routine with AC2 containing the specified datum, and then releases the DD to free memory.

When the delay manager task has processed all possible delay descriptors, it releases DDLOK and does a ?DELAY. If there are more entries in DDCHN, the ?DELAY will remain in effect until the first entry expires (remember, DDCHN is sorted); if DDCHN is empty, the delay is set for one hour (an arbitrary value).

10.2 Data Structures

This section explains the major data bases used by the EXEC. These data bases can be broken down into descriptors, free memory, the in-core queue, the disk queue (file :queue:q), and user profiles.

Descriptors are used to keep track of consoles, batch streams, units, mount requests, coops, and the ?SENDer.

Free memory is used both for building console and coop descriptors and for temporary storage such as a profile during a logon sequence.

The in-core queue is a highly abbreviated version of the disk queue. The in-core queue contains sufficient information to allow entries to be selected from the disk queue with the minimum number of disk reads (nearly always only 1!).

The disk queue is permanent across SYSTEM and EXEC terminations and contains all information necessary to restore the queues to the state they were in when the EXEC was last run and contains all information for all queue requests for which servicing has not been completed.

User profiles are created by the profile editor program, PREDITOR, and are used by the EXEC to determine who can access the system, and how so.

10.2.1 I/O Descriptors

Defined in: XPARS.SR

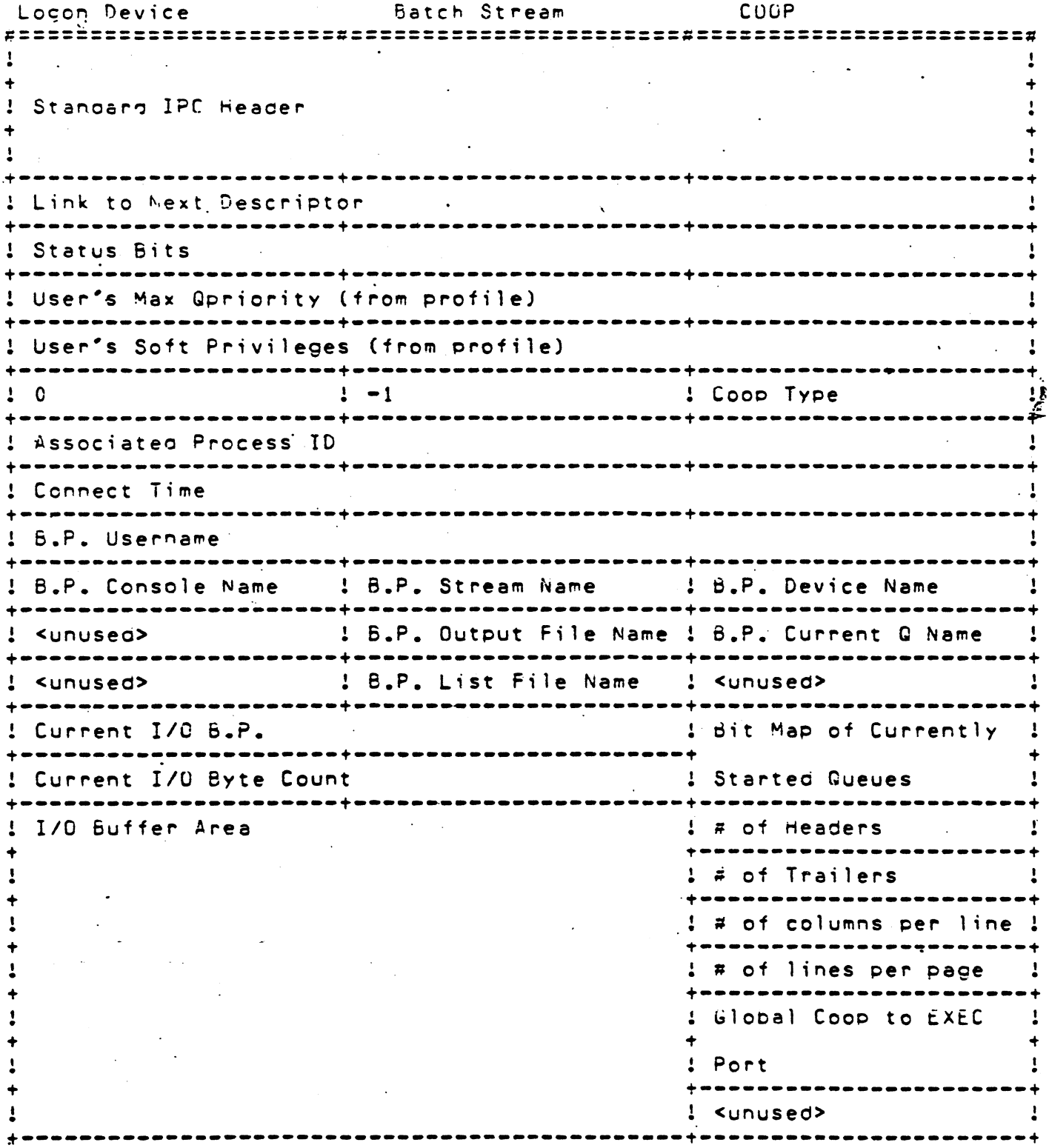
Logon descriptors are created from free memory for each console when it is enabled and released to free memory when that console is disabled. These are called "LD"s and are linked in "LDCHN".

Batch descriptors are assembled in (in EXEC.PR) and always exist. These have no name and are linked in "BSCHN".

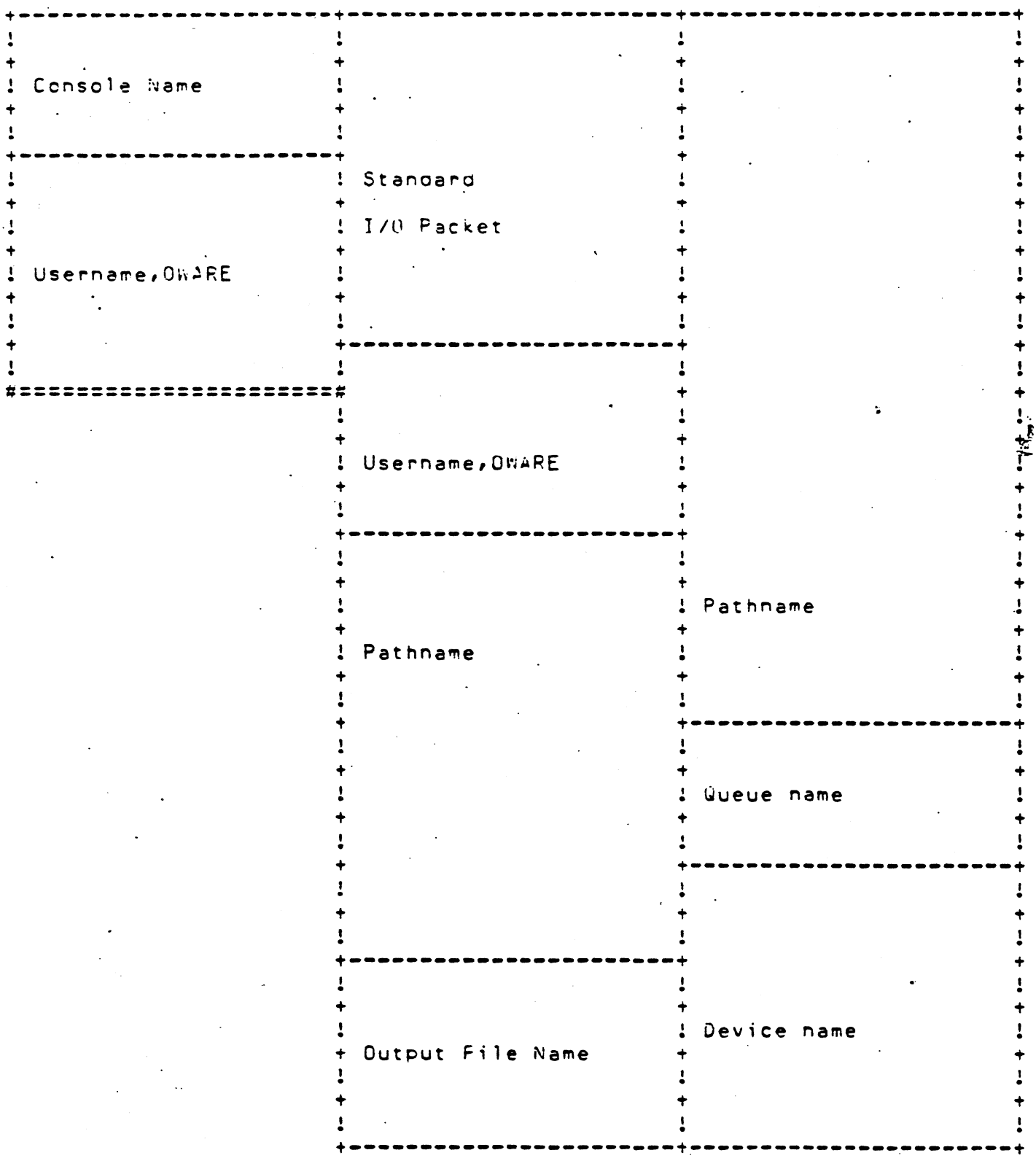
Coop descriptors are created from free memory when a spooled device is started and released to free memory when the device is stopped. These are called "CD"s and are linked in "CDCHN".

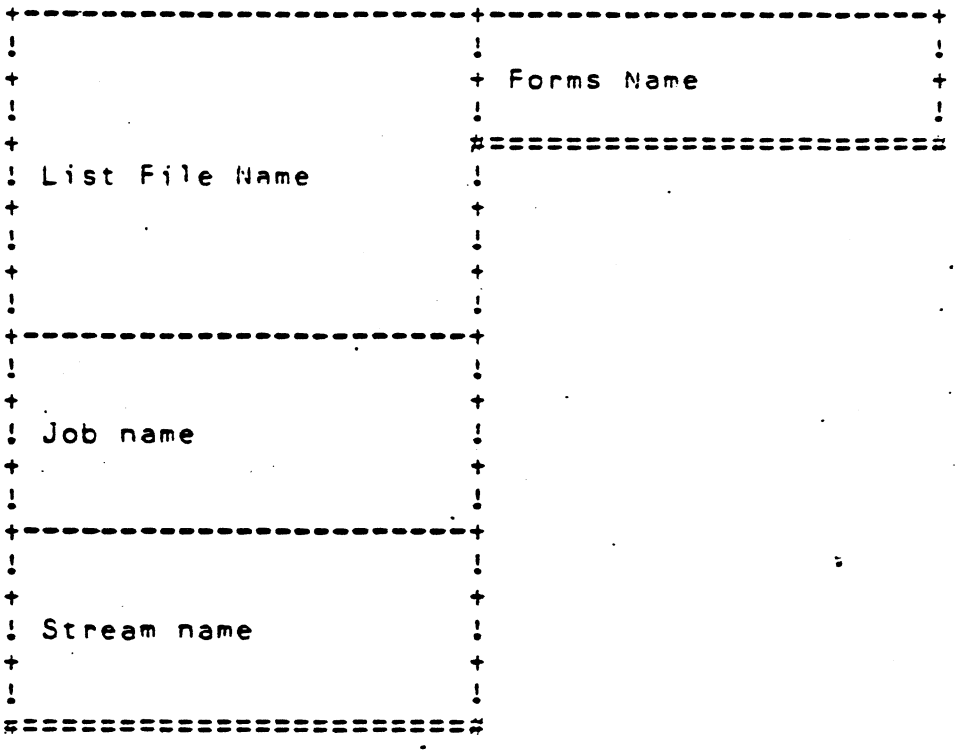
The ?SENDer descriptor is assembled in (in EXEC.SR). There is only one of these. It exists for the life of the EXEC and its address is stored in ZREL location ".OPCN" (operator console). This descriptor is actually a slightly modified LD which is used to generate the "FROM PID 3: (EXEC) ... " messages.

There are areas of commonality among the descriptors. Logon descriptors, batch stream descriptors and the ?SENDer descriptor can all be used with the EXEC's I/O routines (.OPN, .OCHR, .OSTR, .CLS, etc.); batch stream descriptors and coop descriptors can both select and process queue request entries.


Locon Device
Batch Stream
COOP
Standard IPC Header
Link to Next Descriptor
Status Bits
User's Max Priority (from profile)
User's Soft Privileges (from profile)
0
-1
Coop Type
Associated Process ID
Connect Time
B.P. Username
B.P. Console Name
B.P. Stream Name
B.P. Device Name
<unused>
B.P. Output File Name
B.P. Current Q Name
<unused>
B.P. List File Name
<unused>
Current I/O B.P.
Bit Map of Currently
Current I/O Byte Count
Started Queues
I/O Buffer Area
# of Headers
# of Trailers
# of columns per line
# of lines per page
Global Coop to EXEC
Port
<unused>

	Stream number	<unused>
Characteristics	Command word	<unused>
	Stream Lock	<unused>
Control	Status Bits	
Port	Queue Number	
Read	Date Enqueued	
Port	Time Enqueued	
Write	Limit	
Port	Queue Priority	
Stack Base Address	Flags	
Stack Pointer	Sequence Number	
Frame Pointer	B.P. Username	
Stack Limit	B.P. Jobname	B.P. Forms
	B.P. Pathname	
	Core Queue Address	
Return Block	XW0	
	XW1	
	XW2	
WTLD Restart Address	XW3	
Logon Attempt Count	Hi Queue Priority Limit	
Profile Buffer Addr	Lo Queue Priority Limit	
Non-zero for ?SEND	Enqueuer's PID	


Console Name
Standard
I/O Packet
Username,OWARE
Username,OWARE
Username,OWARE
Pathname
Pathname
Queue name
Output File Name
Device name



Notes:

Usernames are always stored in "ACL format" and thus require 18 bytes rather than 16. ACL format is username<0><OWARE><0>.

Device names, console names, and unit names are always stored with a leading @. There is sufficient space reserved to store the @, 31 bytes, and a null for devices and units. However, for consoles, there is space only for the @, 8 bytes, and a null. So console names are limited to eight characters.

The section of the batch and coop descriptors from "queue number" to "enqueuer's PID" is a packet for the EXEC's queue selection routine, ".GSEL".

The "standard I/O packet" in the batch descriptor is used for writing to the jobs output file (via ?OPEN/?WRITE/?CLOSE level I/O).

All three descriptor types begin with a "standard IPC header". For consoles, the header is used to do IPC level I/O with the peripheral manager, for batch streams to send an initial IPC at process creation time, and for coops to send IPC messages to the coop.

Whenever the EXEC finds a queue entry to be processed by an idle and not paused coop, the EXEC sets up the coop descriptor, and ?ISENDs the entire coop descriptor to the coop. This means that whenever the descriptor is altered in any way, all of the coops must be reassembled.

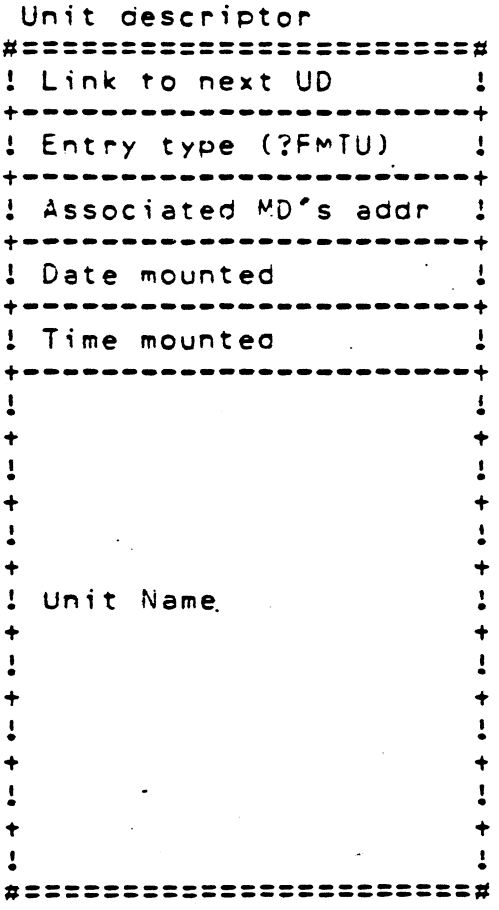
10.2.2 Mount System Descriptors

Defined in: XPARS.SR

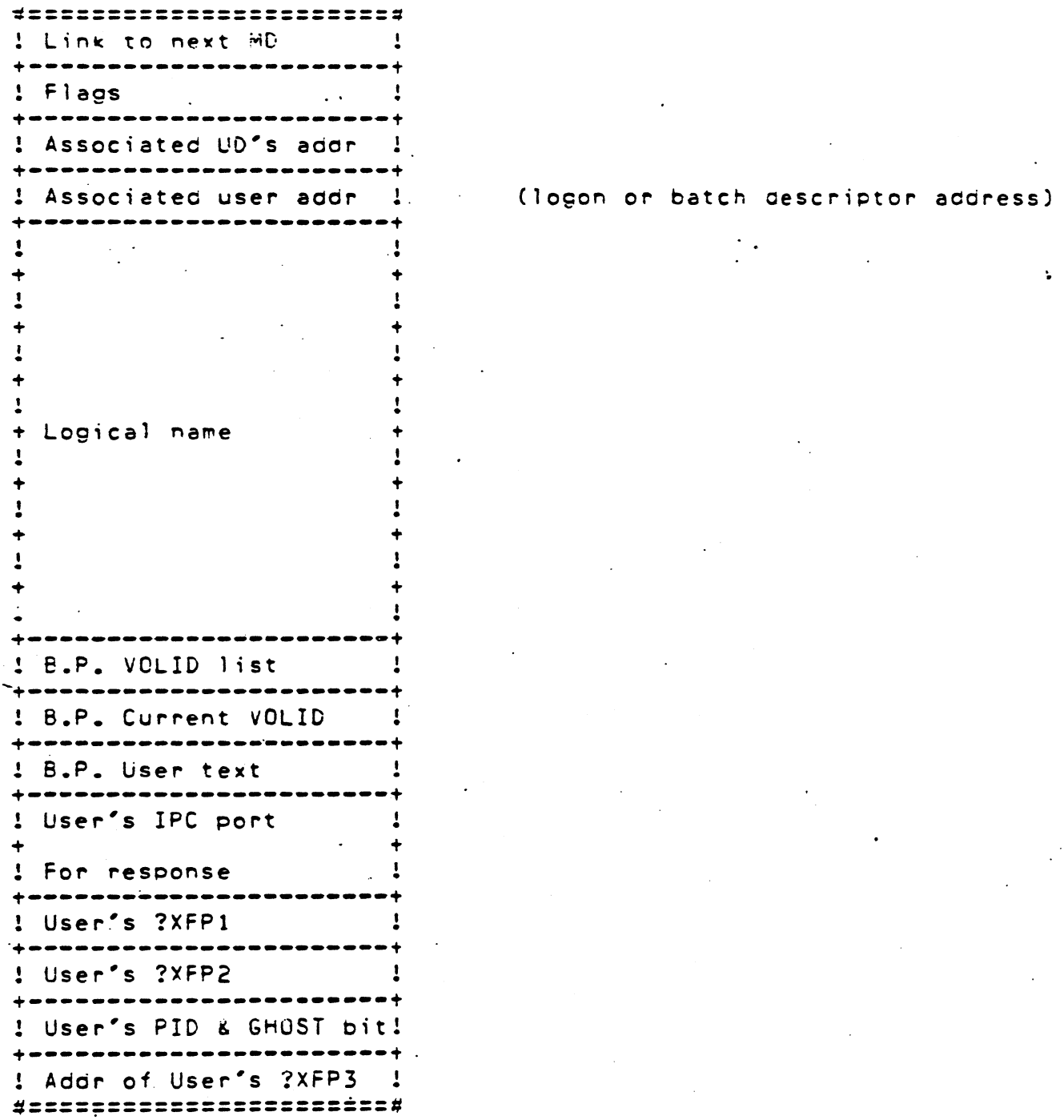
There are two types of descriptors used to handle the mount system, namely: unit descriptors and mount descriptors.

Unit descriptors are created from free memory during EXEC initialization and are never released. EXEC initialization uses ?GNFN (get next filename) to read through the peripheral directory and issues ?FSTAT for each entry it finds looking for magtape units (?FMTU). For each magtape unit it finds, it builds a unit descriptor and chains them together, pointed to by "UDCHN".

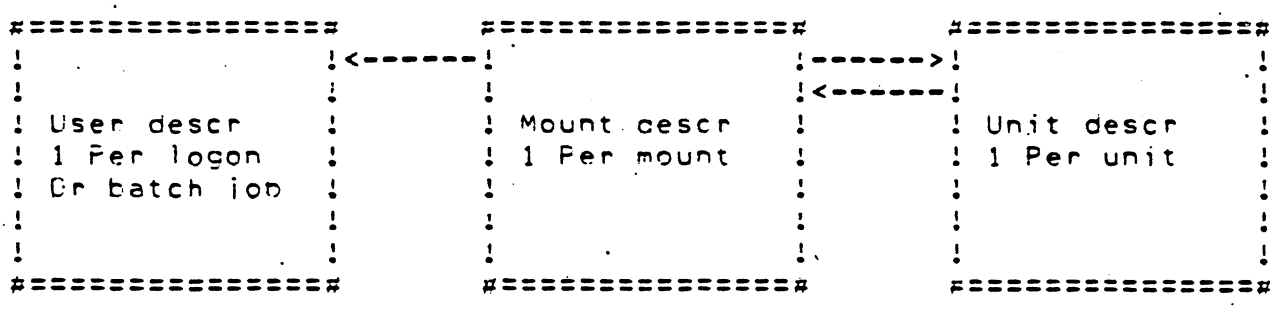
Unit descriptor



Mount descriptors are created from free memory when a mount request (explicit or implicit) is issued by a user and exit until a matching dismount request (explicit or implicit) is issued. Mount descriptors are chained together, pointed to by "MDCHN".



Relationship among MD's, UD's and user descriptors:



10.2.3 Free Memory

The free memory pool extends from the EXEC's initial ?NMAX to the end of the last memory page allocated to the EXEC. The EXEC will issue the ?MEMI system call whenever it needs to expand the pool, but will never shrink the pool.

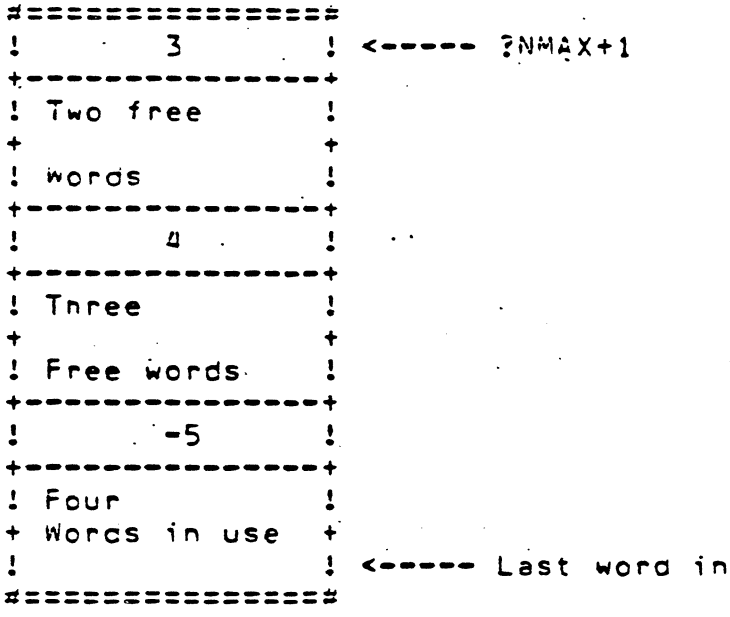
The pool is designed so that any length area can be acquired, and also that the acquirer need not supply the length of the area at release time.

Free space is garbage collected at acquire time, but only insofar as it is required to find the first free area large enough to satisfy the acquirer. The first part of the area so found is given to the acquirer, and the remainder is left free.

The pool is organized as a series of areas, each beginning with its length. If an area is free, the length is a positive value, if in use, the length is a negative value. The length word is "invisible" to the acquirer. When n words are requested, n+1 are found, -(n+1) is stored in the first word, and the requestor is given the address of the second word.

At release time, the releaser specifies the area's address, the release routine validity checks the word before the given address to make sure it is negative, and sets it positive to indicate that the area is free.

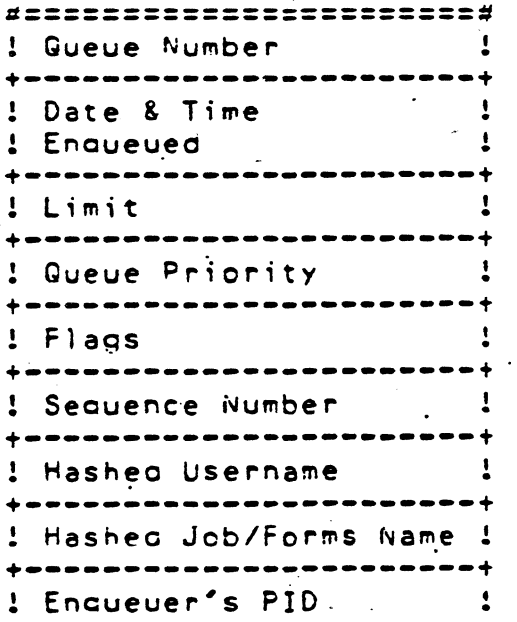
Example:



10.2.4 The In-core Queue

Defined in: XQPARS.SR

The in-core queue is assembled in (in EXQUE.SR) and contains 256 entries. All entries are the same length (CQELTH). The queue is scanned by inspecting every CQELTH word. A non-zero value indicates that the entry is in use and specifies the number of the queue for which that entry is a member. When the sign bit (100) is set, the entry is already being processed. The base address of the in-core queue is at address "CQ" and its entire length is expressed by the symbol "CQLTH".



This timestamp is a double precision number of two second intervals since 31-dec-67, the beginning of time.

There are also two 32 word tables assembled in (in EXQUE.SR) which are used for the queue names and the status of each. "QNAMS" is a table (indexed by queue number) of byte pointers to queue names. "QSTATS" is a table (indexed by queue number) of status words. The left byte of an entry in QSTATS contains flags (queue is open, queue may not be "OPEN"ed, queue may not be "START"ed, queue may not be "DELETE"ed) and the right byte contains the queue's type. Queue types are batch, print, plot, and purch.

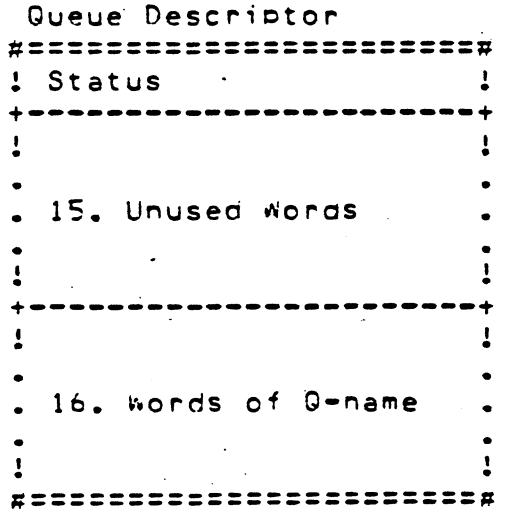
10.2.5 The Disk Queue

Defined in: XQPARS.SR

The disk queue file is ":QUEUE:Q" and is always present. The EXEC will always re-create it at initialization time if necessary. The EXEC also always sets the disk queue's ACL to "+ R" so that all users can use the CLI "QDISPLAY" command.

The CLI "QDISPLAY" command reads the disk queue directly from the disk with no intervention from the EXEC. As a result, one module (ZQUE.SR) of the CLI must be assembled with EXEC parameter file XQPARS.SR in addition to the CLI's own parameter file(s). Whenever any disk queue parameters are changed, CLI must be reassembled with ZQUE.SR and BINDed as a new CLI.

The disk queue file has two sections. The first 4 disk blocks contain 32. queue descriptors, each 32. words long. The remaining blocks (up to 256. of them) each can contain one queue entry.



max queues = 32
max entries = 256

Note that queues are numbered 1-32, not 0-31, so the zeroth queue descriptor is for queue number 1, and so forth.

Queue entry

Status	Same as in-core queue
Date Enqueued	File system format
Time Enqueued	File system format
Limit	Same as in-core queue
Queue Priority	Same as in-core queue
Flags	Same as in-core queue
Sequence Number	Same as in-core queue
XW0	
XW1	
XW2	
XW3	
/AFTER Date	File system format
/AFTER Time	File system format
89. Unused words	
8. words of user name	
16. Words of JOB/FORMS Name	
64. Words of pathname	
64. Unused words	

10.2.6 User Profiles

Defined in: PRPARS.SR

A user profile is a one block disk file in directory :UPD and has a filename the same as the username it profiles. For example, Tocp's user profile would be ":UPD:TOCP". User profiles are created by PREDITOR. :UPD is also created by PREDITOR if necessary. The EXEC reads a user's profile when he attempts to log on either at a console or in a batch stream, or when he attempts to submit a batch job through the stacker coop.

Each profile has in it a profile-format-rev number which is checked both by the EXEC and the PREDITOR to make sure that they understand that particular profile's format.

Whenever changes to any parameters in PRPARS.SR occur, it is necessary to reassemble both the EXEC and the PREDITOR.

User Profile	
Profile-Format-Rev	
8. Words of password	
32. Words of initial Program pathname	
32. Words of initial Message file pathname	
?PROC privileges	
?PROC max sons	
?PROC priority	
?PROC max memory	
Maximum number of Disk blocks	
Usage privileges	Such as batch allowed, console allowed
Last logon time	?GTOD format
Last logon date	?GDAY format
Max queue priority	
169. Unused words	

10.3 EXEC Initialization

1. Get father's PID and store it in DADDY.
2. Turn SUPERUSER and SUPERPROCESS on.
3. Get the username and store it in OPNAM (operator name).
4. Set up the memory pool.
5. Insure that the initial working directory is :PER (if not abort).
6. Verify that there is no other EXEC running by looking for the IPC files EXEC needs. (if they are there, abort).
7. Create the IPC ports (EXEC, EXEC_REQUEST, LMT).
8. If there is no :UDD, create it, and set up its ACL.
9. If there is no queue directory, create it and set up its ACL.
10. Set ACL on LMT.
11. Get system rev number and store it in AOSREV.

INIT in XIOV0

INIT1 in XIOV1

12. Get full pathname of current EXEC to determine which directory this EXEC was proc'ed from so we can insure using the right COCPs and messages.
13. Generate pathname to LOGON.MESSAGE, STACKER.PR, XHAMLET.BRK, and the spooler .PR files using the information obtained from step 12.
14. Start up the other tasks.
15. Let the operator know we are ready.
16. Initialize the queue manager.
    a. Create :QUEUE:Q (if it exists, use the old one).
    b. Clear the in core queue.
    c. Read in the disk queue and obtain the highest sequence number in use, pend any /AFTER type entries, mark jobs that died in action as needing restart.
    d. Rebuild the queue name and queue status tables.
    e. Unlock the queue.
17. Delete the mount queue (just in case - rev 1 used to leave it around)
18. Set ACL for :QUEUE:Q.

19. Re-open any queues that were open when we last shut down.

INIT1 in XIOv1

INIT2 in XIOv2

20. If any consoles were specified on the PROC line, enable them.

21. Build the tape unit list.

22. Initialization is done ... poof ... we are now the log on task.

10.4 XLPT

10.4.1 General information

XLPT has 3 tasks:

Input task	(ITSK)
Output task	(OTSK)
Control task	(CTSK)

The code for XLPT is contained in four modules:

XLPT	--	ITSK and CTSK.
XLPT1	--	Misc. subroutines, IPC header definitions, misc. variables and string definitions.
XLPT2	--	more subroutines, variables and strings.
XLPT3	--	OTSK, buffers for messages to PMGR, elongate routines for LP2, UDA temporary storage areas.

Error handling:

.AO	--	Abort with error message sent to EXEC. (enter with error code in AC0)
.RO	--	Send error message to EXEC but do not abort (AC0 = error code)

If an error occurs that is to be reported on the output device and no form is loaded, XLPT will load the default form for the device and print the message. If a form has been loaded, then XLPT simply prints the message.

Simulation:

non standard characters

There is a routine in XLPT2 that is called whenever XLPT encounters a non-standard character (ie. control char, new-line, formfeed). If the simulation flag (SMVFU - set at XLPT initialization time) is not set, the routine just passes the special character along, else it will trap the special character and send a sequence of characters that will simulate the required operation.

10.4.2 XLPT initialization

1. Get the initial message containing ports for communication with EXEC.
2. Start up the output task (OTSK in XLPT3) and wait for it to signal that it has opened the print device, and decided what type of device it is (PMGR controlled or not).
3. Start up the control task (CTSK in XLPT).
4. Generate the AOS and XLPT revision numbers and AOS SYSID string.
5. Send "Cooperative intialized" message to EXEC.
6. Send "Idle" message to EXEC.

At 2 above, OTSK was started. During initialization, it performs the following:

1. Open the print device, and get device type.
2. If device is non-datachannel (ie. needs the PMGR)
   a. Set the SMVFU flag to non-zero
   b. Set the PMGRF flag to indicate that we will do ISENDs to the PMGR for output.
   c. Signal the input taks (which is waiting for us) that PMGRF has been set.
   d. Get port information so that XLPT can do primitive IPCs to PMGR.
3. If device is datachannel If LPB (VFU available), then enable VFU formatting, and become resident. If not LPB, set simulation flag (SMVFU).
4. Send waiting for a buffer to be enqueued to the readied buffer queue.

At this point, XLPT is initialized and ready to receive print requests from EXEC.

10.4.3 Normal XLPT operations

10.4.3.1 CTSK -- the control task

Module: XLPT Function: receive IPC messages from EXEC and set appropriate flags for desired operations.

C1. ?IREC for message from EXEC.

C2. Dispatch on command code as follows:

Code	Brief description	Action
1	Process file	wake up ITSK (?XMTW) then go to step C1.
2	Flush current file	set location FLUSH to -1 and jump to step C1.
3	Shut down	set location DEATH to -1, start up OTSK, and go to step C1.
4	Restart	set location RSTRT to -1 and go to step C1.

else call .BOMB.

10.4.3.2 OTSK -- the output task

Module: XLPT3 Function: perform the actual output operation by doing either IPCs to the PMGR or ?WRBs to the disk or data channel printer.

O1. Wait for a buffer to be ready for output.

O2. If the output device is controlled by the PMGR (PMGRF<>0) then send the information in the buffer using primitive IPCs.

O3. If the output device is not controlled by if the PMGR (PMGR=0) then:

a. If the buffer is only partially full, pad with nulls to 512. bytes.
b. Perform a WRB to the output device.
c. ISZ the double precision block number value in the WRB packet

O4. Go back to step O1.

10.4.3.3 ITSK -- the input task

Module: XLPT Function: perform file opening and reading, and VFU simulation if necessary.

I1. wait for the control task to signal that there is something to work on.

I2. Step the character (0-9) for the header line.

I3. Set begin and end page number

I4. Clear the following flags:

PRINT -- for supression
FLUSH -- flush current print job
RSTRT -- restart current print job
DAPSW -- /DELETE option requested
FSPKT+?STCH -- ?FSTAT flag (FSTAT already run)

I5. Check for user read access to the file. If not then error (see below)

I6. Open file (on error goto STERR [see below]).

I7. Perform FSTAT on file.

I8. See if /FORMS switch is present. If so, read and validate the FORM file's UDA (there is checksum) and if all is okay, set .WORK to indicate the form to use.

I9. If no /FORM then see if the user file has a valid UDA. If so, make sure LPP matches the current LPP and that the CPL < the current CPL. (If not ... error) Set .WORK to indicate a user defined form.

I10. If there is no /FORM or user form then use the default form.

I11. Set up the in core constants (width, lpp, top of form line ...).

I12. Load the VFU. (the routine is a no-op if non-VFU device).

I13. Set up tab information.

I14. Set device characteristics.

I15. Set elongation (if LP2)

I16. Check the following flags:

a. Cancelled by operator - print notification message, close the input file, print trailer, and start next request (step I1.)
b. Cancelled by user - same as Cancelled by operator
c. Not restartable - if the file is restarting, and this flag is set, then abort.

I17. Print out header(s).

I18. Set up page limit (if requested).

I19 Position to beginning of file.

I20. Make sure we are at Top of Form (TOF).

I21. If this is the 2nd or later copy, see if an extra TOF is necessary to insure an even page count.

I22. Perform a ?RDB on the input file.

I23. If error on RDB:

a. If EOF then set FEOF and continue.
b. If any other error, then pass the error code back to EXEC.

I24. Move characters from the single input buffer to the output buffer. (OTSK) will pick up the output buffer and print it.

Note: This is where XLPT interprets characters. If VFU simulation is necessary, then the characters needed to do the simulation are moved into the output buffer (as opposed to the VFU control character) (see .SIMUL in XLPT2).

I25. See if any errors occurred that would force XLPT to abort (ie. illegal VFU character, page limit exceeded, etc.) If so, print the error message.

I26. See if FEOF(end of file flag) is set. If so skip to step I30.

I27. See if CTSK has set the force flush flag (FLUSH). If so, print message and abort.

I28. See if CTSK has set the restart flag (RSTRT). If so, print message, output trailer to make page count even (if necessary), close the input file, and start over (go back to step I1).

I29. ISZ current input file block pointer and goto I22.

I30. If more copies are needed, increment copy counter and goto step I19.

I31. Close the input file.

I32. If /DELETE option was requested, verify that the user has the right to do so (owner of the file and write access to the directory). If the user does, do it.

I33. Output any required trailer.

I34. Goto I1. (start again)

THIS PAGE ORIGINALLY WAS LEFT BLANK

CHAPTER 11 - CLI
(updated for AOS Rev. 3.11)

11.1 Introduction

The general topics discussed in this chapter will deal with the CLI's structure and command processing. An example will be used to explain some basics regarding the operation of CLI.

11.2 The Structure

11.2.1 Overview

The structure of CLI can be broken down into four categories: its tasks, its size, stack and operating environment.

There are three types of tasks - primary, ^C^A interrupt and utility. The primary task of CLI is responsible for issuing a read from @INPUT, transferring the input to stack and editing, expanding any macros, validating switches and arguments with command descriptors and executing the command. The ^C^A interrupt task is responsible for taking care of interrupts from the primary (main) and utility tasks, resetting the stack, list flag, command input pointer and output buffer pointer and issuing error message. The utility task coordinates the handling of I/O buffering.

The sizing of CLI is determined by the number ZREL and NREL locations used. ZREL locations will contain processing mode flags, temporary values and pointers. NREL locations are divided into shared and unshared code. The unshared area contains the buffers, packets, stacks and names. The shared code region contains the reentrant programs.

The stack structure is set up to independently handle each of the tasks of CLI. Additionally, stack space is allocated for CLI initialization.

CLI's environment is made up of databases containing buffers, lists, filenames and flags controlling the various processing modes. Level control through the use of the stack is also part of CLI's environment.

11.2.2 Tasks

Upon initial entry into CLI, initialization takes place. The stack gets initialized. The @INPUT and @OUTPUT packets get opened for the primary (main) task. Initialization packets are invoked for ^C ^A and utility tasks followed by CLI's initial message CLI then enters the primary task.

11.2.2.1 Primary task

The following outline defines the basic steps in CLI's primary task.

1. Type CLI header message
2. Set-up the input buffer
3. Close temporary list file if open
4. Execute prompt commands if any
5. Read line from @INPUT (.ISTR)
6. Push contents of input buffer onto the stack
7. Initialize variables controlled by global switches
8. Check for more lines on input
9. Begin editing the input
10. Check for angle brackets, < >
    if present, expand the brackets and return to step 9, else
11. Check for parentheses, ( ) if present, expand the parentheses and return to step 9, else
12. Check for macro, if present, expand the macro and return to step 9, else
13. Look up the command (.LKUP) if command is unknown add macro brackets and return to step 9, else if the abbreviation is not unique, issue an error message, (.E0) and return to step 5 else
14. Build the switch and/or argument descriptors
15. Verify that argument existence is consistent with commands description, if not, issue an error message (.E0) and return to step 4, else
16. Dispatch to the command
17. Remove the command from the buffer and return to step 7

11.2.2.2 ^C^A Interrupt Task

The function of the ^C^A interrupt task is to interrupt the task control- ling the console. In the area of CLI this may be either the primary or utility task. For the primary task, a ?IDGOTO interrupt is issued by the ^C^A. ?IDGOTO is disabled for the utility task. In both cases, the stack is reset together with the list flag, command input pointer and output buffer pointer. An error message is then issued before CLI goes to get another line from @INPUT.

11.2.2.3 Utility Task

The primary function of the utility task is to handle double buffering for loads and dumps. This is accomplished through a JMP 0, MESSAGE where MESSAGE contains the address for a read or write of the other buffer.

11.2.3 Size

The following defines the size constraints on CLI.
ZREL locations - ?ZMAX=374; this area contains processing mode flags, temporaries and indirect addresses.
NREL locations - ?NMAX=1677; the unshared code area (approximately 500 words to start with) contains buffers, packets, stacks and names. The shared code area occupies the area (34000-77271).

11.2.4 Stack Structure

Initialization - when CLI begins execution stack requirement is 30.words in unshared NREL.

Primary - begins at ?NMAX; extended by ?MEMI calls when stack fault occurs. If the ?MEMI fails, the stack is reset to ?NMAX, the stack fault is set to abort, and a ?CHAIN is made to the same CLI.

^C^A - uses 20.words in unshared NREL.

Utility - uses 20.words in unshared NREL.

11.2.5 Environment

This section is a detailed breakdown of AOS's operating environment. The environment consists of databases and levels. The data base is made up of buffers and flags. The following describes the buffers and their use in the database.

```
IBUF   - the input buffer
         size: 128 bytes  (with screenedit off)
         size:  76 bytes  (with screenedit on)
BUF    - the output buffer
         size: 133 bytes
STRING - current buffer
         size: 128 bytes
```

Also, there are a series of flags used to control CLI I/O and maintain CLI status. The most important ones are:

BCMODE	-1 for batch on 0 for console on 1 for son of exec
DINI	-1 for @DATA or byte pointer to data filename
FRAME	frame pointer at start of command execution
GOCHNS	?GOPENED channel bit map
LINI	-1 for @LIST or byte pointer to list file name
LSTIT	0 for all or address of list output packet non-zero if command changes directories
OCHNS	?OPENED channel bit map
SCGBL	total number of system calls made
SCLCL	number of system calls since last performance command
SCRMOD	screenedit mode 1: off -1: on
SPPMOD	superprocess mode 1: off -1: on
SUPMOD	superuser mode 1: off -1: on
EMODE	class 1 error mode QMODE squeeze mode
WMODE	class 2 error mode

Levels in CLI's environment are maintained and controlled by copies of the prompt list, string, datafile name, listfile name, search list, working directory, home pointer previous top of stack and "new top" of stack and a set of processing mode flags. The buffers maintained for level control are:

DIRBUF	contains the working directory at start of previous commands processing size: 128 bytes
PRMLST	the prompt list buffer size: 9 words
LNAM	list file name size: 128 bytes
DNAM	datafile name size: 128 bytes

The flags used for level control are:

BLV1 - level at last input
LEVEL - environment
TLVL - level last typed out

11.3 Command Processing

11.3.1 Sequence of Operations

The CLI begins by reading a line from @INPUT and pushing it onto the stack. Each character is examined for angle brackets, "< >", parentheses, "( )", square brackets, "[ ]", and the null at the end of the command.

The expansion of angle brackets will cause the previous "word" to be repeated for each argument in the brackets while parentheses expansion will cause the entire command to be repeated for each argument. The use of square brackets will cause the macro to be expanded into the command space on the stack.

The command string is then edited. That is, the initial spaces are removed and multiple spaces and/or tabs are compacted into single commas. This puts the CLI command into a standard form with fields delimited by commas.

The command is then isolated and looked up in the command table (CTBL). When abbreviations are used, the entire command table is searched to insure that there are no multiple matches.

Switches and arguments associated with the various commands are validated with respect to the command descriptor. As in the case of the commands, unique abbreviations for switches and arguments are accepted. An error is returned when there is a multiple switch.

The command is actually executed by being dispatched through a table via a JSR.

Commands remaining in the input buffer are shifted to the beginning of the buffer and processing continues.

11.3.2 Macro Processing

The macro file is first opened and read. Each character is then examined and pushed onto the stack. When a delimiter is found in the command string namely a null, new line, formfeed, or carriage return, the next character is checked for an ampersand (&). Whenever an ampersand is found neither the ampersand nor the delimiter will be pushed on the stack.

When a percent sign (%) is encountered the dummy argument is replaced with the calling argument. If the dummy argument is % n / ... all the switches on the argument in the call are searched for every / on the argument in the dummy.

When the dummy argument is %/n\... all the switches on the argument in the dummy are searched for every \ on the argument in the call.

11.3.3 Command Data Bases

11.3.3.1 Command Table Data

The following defines the Command Table entry format for both commands and pseudo macros.

```
entry0 : byte pointer to ASCII string for command or pseudo macro
entry1 : command entry point
entry2 : 0
```

This table is found in module ZDISP

11.3.3.2 Switch Descriptors

Commands with simple switches:

```
-2 : A-P switch bit map (all 16 bits)
-1 : Q-Z switch bit map (bits 0-9)
     and argument flags (bits 10-15)
 0 : command entry point
```

Commands with complex switches:

switch table entry

```
entry 0 : byte pointer to ASCII string for switch
entry 1 : address of routine for switch without a value: 0 if the
          switch takes a value
entry 2 : 0 if the switch does not take a value; address of
          routine for switch with a value

     -2 : address of switch table
     -1 : argument flags (bits 10 - 15)
```

11.3.4 Stack Structure for Command Calls

11.3.4.1 Commands with Simple Switches (single character)

1. Command line as input <null>
2. Command <null>
3. SW1/SW2.../SWn<null>
4. Argument 1 <null>

   .
   .
   .
   .

5. Byte pointer to argument 1

   .
   .
   .

6. Argument count
7. A-P switch bit map
8. Q-Z switch bit map

11.3.4.2 Commands with Complex Switches

Complex Switches (multi-character)

1. Command line as input <null>
2. Command <null>
3. Switch 1 <null>
4. Argument 1 <null>
   .
   .
   .
   Argument n<null>
5. Address of switch 1 routine
6. Byte pointer to switch 1 value or 0)
7. Switch count
8. Byte pointer to argument 1
9. Argument count
10. 0
11. 0

11.4 Examples

11.4.1 The CLI PUSH Command

Module Name: ZCMD4
Entry Point: XPSH

First, ^C^A interrupts are disabled (.PIG). Seventeen (17) words of stack are then allocated and then filled with a word pointer to each one of the environment parameters. The contents of the parameters are then pushed on top of the pointers in the following order after which ^C^A is enabled:

1. The squeeze mode status
2. The CLASS2 mistake reaction
3. The CLASS1 mistake section
4. The prompt list
5. The string variable
6. The data file status and name
7. The list file status and name
8. The searchlist status and name
9. The working directory
10. The screenedit state
11. The superprocess state
12. The frame pointer
13. The superuser state
14. The beginning of the pointer block
15. The next available address after the parameters

11.4.2 The CLI POP Command

Module Name: XCMD4
Entry Point: XPOP

The ^C^A interrupts are disabled, the inverse of PUSH is performed and ^C^A enabled.

11.5 Template Expansion

Module Name: ZTEX

Several commands, namely: DELETE, FILESTATUS, TYPE, LOAD, DUMP, and MOVE use the template expander. The expander is a collection of subroutines that breakdown the input into modes, that is the pieces separated by colons and then search the directories encountered for files that match the template.

11.6 CLI Module Names

CLI module names and a description of their contents.

Main modules	Contents
Z	contains the primary task, ^C^A task and utility task
ZAWE	processes: warning and error messages
ZDISP	contains the command dispatch table
ZCMD0	bridges: HELP, PROMPT, MESSAGE, PERFORMANCE, DATE, TIME SYSLOG processes: BYE, WRITE
ZCMD1	bridges: HOST, WHO, SQUEEZE, PRTYPE, PRIORITY, BLOCK, UNBLOCK, TREE, RUNTIME processes: TERMINATE
ZCMD2	processes: PROCESS, CHAIN, EXECUTE, XEQ, DEBUG, CHECKTERMS
ZCMD3	bridges: STRING, DATAFILE, LISTFILE processes: CHARACTERISTICS
ZCMD4	bridges: CLASS1, CLASS2 processes: PUSH, POP, LEVEL, PREVIOUS, CURRENT
ZENV	bridges: SEARCHLIST, DIRECTORY, processes: !SEARCHLIST, !DIRECTORY, !PATHNAME, !USERNAME

ZFIL	processes: FILESTATUS
ZFMC0	bridges: DELETE, RENAME, TYPE processes: CREATE, COPY
ZFMC1	bridges: REVISION, SPACE, ACL, DEFACL processes: !DEFACL, !ACL
ZFMC2	bridges: ENQUEUE, PATHNAME PERMANENCE
ZMATH	contains set-up for all arithmetic routines, pseudo macro !OCTAL
ZMOLD	processes: DUMP, LOAD, MOVE
ZOP	bridges: INITIALIZE, BIAS, PAUSE, REWIND, ASSIGN, DEASSIGN processes: RELEASE, CONTROL, SEND
ZOV1	processes: DATA, LIST, HELP, MESSAGE, PERFORMANCE
ZOV2	processes: BIAS, BLOCK, CLASS1, CLASS2, HOST, INITIALIZE PAUSE, PRIORITY, PRTYPE, RUNTIME, SCREENEDIT SQUEEZE, SUPERPROCESS, SUPERUSER, STRING, SYSID, SYSLOG, TREE, UNBLOCK, WHO
ZOV3	processes: ACL, DATE, DEFACL, DIRECTORY, ENQUEUE, PATHNAME, PERMANENCE, PROMPT, REVISION, SEARCHLIST, SPACE, TIME
ZOV4	processes: ASSIGN, DEASSIGN, DELETE, RENAME, REWIND, TYPE
ZPARS	contains the parameter file for CLI
ZPSM	processes: !EQUAL, !NEQUAL, !END, !ELSE, !READ, !PID, !HID, !STRING, !LOGUN, !OPERATOR, !DATE, !TIME, !ASCII, !EXPLODE
ZQUE	processes: QDISPLAY, QBATCH, QSUBMIT, QPRINT, QPUNCH, QPLOT, QHOLD
ZSTK	handles stack manipulation for CLI
ZSUB	contains common subroutines used by CLI modules
ZTEX	performs template expansion for CLI

Support modules	Contents
TTSK	contains task support for ^C^A
TXMT	supports inter-task communications

11.7 CLI Commands and their modules

CLI commands and their corresponding module names and module entry points. XXX -> YYY indicates that the entry point is actually a bridge to an overlay entry point. [overlay entry point] = [module Entry Point.] For example, the overlay entry point for ACL is XACL<oct>.

Command Name	Module Name	Module Entry Point
ACL	ZFMC1 -> ZOV3	XACL
ASSIGN	ZOP -> ZOV4	XASS
BIAS	ZOP -> ZOV2	XBIA
BLOCK	ZCMD1 -> ZOV2	XBLK
BYE	ZCMD0	XBYE
CHAIN	ZCMD2	XCHN
CHARACTERISTICS	ZCMD3	XCHA
CHECKTERMS	ZCMD2	XLSN
CLASS1	ZCMD4 -> ZOV2	XCL1
CLASS2	ZCMD4 -> ZOV2	XCL2
CONTROL	ZOP	XCON
COPY	ZFMC0	XCOP
CREATE	ZFMC0	XCRE
CURRENT	ZCMD4	XCUR
DATAFILE	ZCMD3 -> ZOV1	XDTA
DATE	ZCMD0 -> ZOV3	XDAT
DEASSIGN	ZOP -> ZOV4	XDEA
DEBUG	ZCMD2	XDEB
DEFACL	ZFMC1 -> ZOV3	XDEF
DELETE	ZFMC0 -> ZOV4	XDEL
DIRECTORY	ZENV -> ZOV3	XDIR
DISMOUNT	ZQUE	XDIS
DUMP	ZMOLD	XDUM
ENQUEUE	ZFMC2 -> ZOV3	XENQ
EXECUTE	ZCMD2	XEXE
FILESTATUS	ZFIL	XFIL
HELP	ZCMD0 -> ZOV1	XHLP
HOST	ZCMD1 -> ZOV2	XHST
INITIALIZE	ZOP -> ZOV2	XINI
LEVEL	ZCMD4	XLVL
LISTFILE	ZCMD3 -> ZOV1	XLIS
LOAD	ZMOLD	XLOA
MESSAGE	ZCMD0 -> ZOV1	XMES
MOUNT	ZQUE	XMOU
MOVE	ZMOLD	XMOV
PATHNAME	ZFMC2 -> ZOV3	XPAT
PAUSE	ZOP -> ZOV2	XPAU
PERFORMANCE	ZCMD0 -> ZOV1	XPER
PERMANENCE	ZFMC2 -> ZOV3	XPMN
POP	ZCMD4	XPOP
PREVIOUS	ZCMD4	XPRE
PRIORITY	ZCMD1 -> ZOV2	XPPR

PROCESS	ZCMD2		XPRO
PROMPT	ZCMD0		XPRM
PRTYPE	ZCMD1	-> ZOV2	XPTY
PUSH	ZCMD4		XPSH
QBATCH	ZQUE		XQBA
QCANCEL	ZQUE		XQCA
QDISPLAY	ZQUE		XQDI
QHOLD	ZQUE		XQHO
QPLOT	ZQUE		XQPL
QPRINT	ZQUE		XQPR
QPUNCH	ZQUE		XQPU
QSUBMIT	ZQUE		XQSU
QUNHOLD	ZQUE		XQUN
RELEASE	ZOP		XREL
RENAME	ZFMC0	-> ZOV4	XREN
REVISION	ZFMC1	-> ZOV3	XREV
REWIND	ZOP	-> ZOV4	XREW
RUNTIME	ZCMD1	-> ZOV2	XRNT
SCREENEDIT	ZCMD1	-> ZOV2	XSCR
SEARCHLIST	ZENV	-> ZOV3	XSEA
SEND	ZOP		XSEN
SPACE	ZFMC1	-> ZOV3	XSPA
SQUEEZE	ZCMD1	-> ZOV2	XSQU
STRING	ZCMD3	-> ZOV2	XSTR
SUPERPROCESS	ZCMD1	-> ZOV2	XSPR
SUPERUSER	ZCMD1	-> ZOV2	XSUP
SYSID	ZCMD0	-> ZOV2	XSID
SYSLOG	ZCMD0	-> ZOV2	XSYS
TERMINATE	ZCMD1		XTER
TIME	ZCMD0	-> ZOV3	XTIM
TREE	ZCMD1	-> ZOV2	XTRE
TYPE	ZFMC0	-> ZOV4	XTYP
UNBLOCK	ZCMD1	-> ZOV2	XUNB
WHO	ZCMD1	-> ZOV2	XWHO
WRITE	ZCMD0		XWRI
XEQ	ZCMD2		XRUN

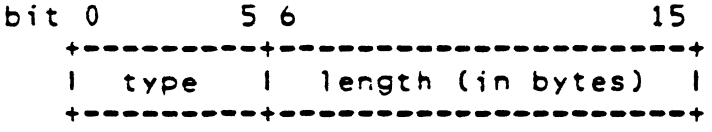
CLI Pseudo macros and their corresponding module names and module entry points.

Pseudo Macro Name	Module Name	Module Entry Point
!ACL	ZFMC1	YACL
!ASCII	ZPSM	YASC
!DATE	ZPSM	YDAT
!DECIMAL	ZMATH	YDEC
!DEFACL	ZFMC1	YDEF
!DIRECTORY	SENV	YDIR
!ELSE	ZPSM	YELS
!END	ZPSM	YEND

!EQUAL	ZPSM	YEQU
!EXPLODE	ZPSM	YEXP
!FILENAMES	ZTEX	YFIL
!HID	ZPSM	YHID
!LOGON	ZPSM	YLOG
!NEQUAL	ZPSM	YNEQ
!OCTAL	ZMATH	YOCT
!OPERATOR	ZPSM	YOPR
!PATHNAME	ZENV	YPAT
!PID	ZPSM	YPID
!READ	ZPSM	YREA
!SEARCHLIST	ZENV	YSEA
!STRING	ZPSM	YSTR
!TIME	ZPSM	YTIM
!USERNAME	ZENV	YUSE

11.8 AOS Dump Format

A dump file created by AOS CLI consists of variable length records, each having a fixed format header containing the block's type and length. The format of the header is:



These blocks are not word-aligned, since the byte length of any of the blocks might be odd. The length refers to the length without the header. Specific block types are detailed below. There is one anomaly to the blocking scheme -- data from data files are not contained in some type of block, but rather follow data header blocks (block type 7 -- see below).

0 -- Start of dump
==========

format:

type: 0	length: 14
dump format revision: 15	
time of dump: seconds	
minutes	
hours	
date of dump: day	
month	
year	

(all numbers are base 10)

may occur only once, at the start of the dump file

1 -- file status block
==========

format:

type: 1	length: ?slth*2
?fstat packet for the file	

This is the first you'll see of a file. It contains useful information like the file type and length. It is always followed by a name block.

2 -- Name block
==========

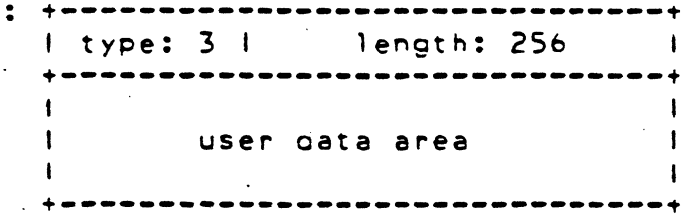
format:

type: 2	length: variable
file name (with null terminator)	

the file name is a simple name (not a pathname). You have to look at the directory start and end blocks to tell what the pathname is.

3 -- Uca block

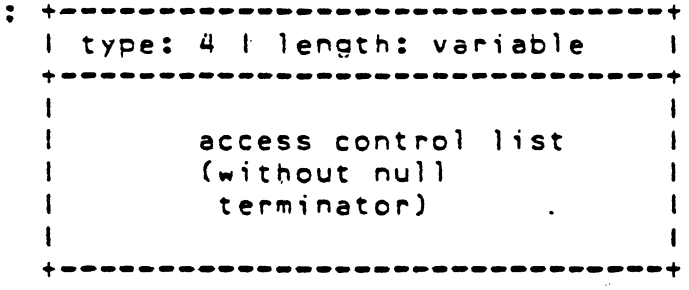
format:



only directories and data files may have uda blocks, and in both cases, it directly follows the end block. Most files do not have a uda -- currently they are used only by infos.

4 -- Acl block

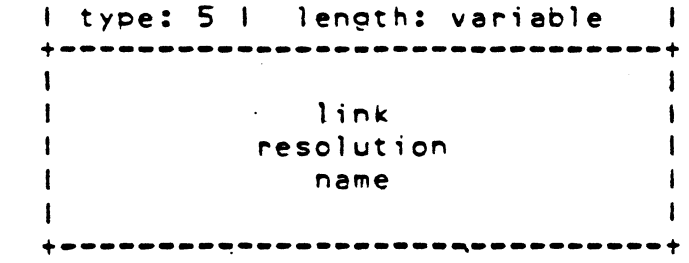
format:



the acl block is used only for directories and data files, and in both cases, it comes right before the end block. The acl block is not always present, and if it is not there, the acl should be set to the user's default.

5 -- Link block

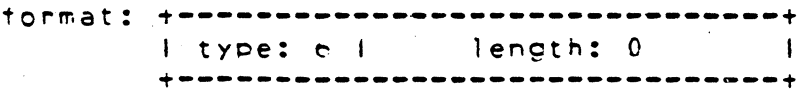
format:



the link block must be there for links, and it follows the name block.

6 -- Start block
==========

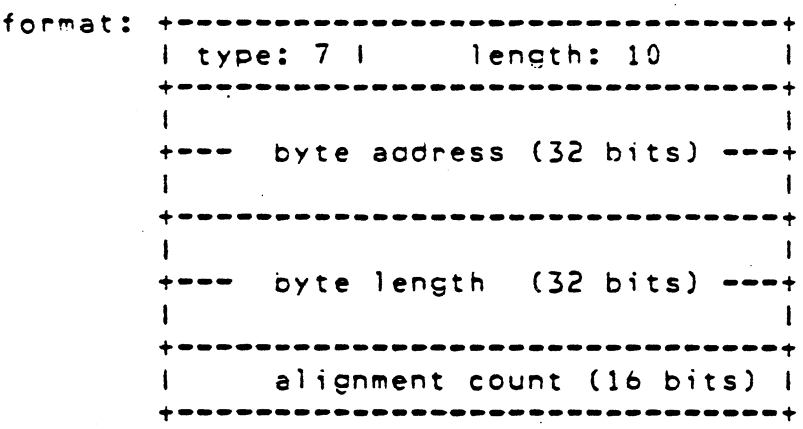
format:



the start block is always present for data files and directories, and is always matched by an end block. It follows the file name block, and after it come data blocks (for data files) or file status blocks of subordinate files (for directories).

7 -- Data header block
==========

format:



the byte address specifies where in the data file to put the following chunk of data. Currently, it must be a multiple of 512 bytes (one disk block). Minus one as the address means continue from where the last block left off (this convention is not now used). In general, hunks may be left out of the data (if one data block does not take up where the previous one left off) -- this happens if all the intervening area was full of zeros.

A data header block will be followed by <alignment count> bytes to be ignored (currently either zero or one), followed by <byte length> bytes of data. Next comes another data block, or an acl block, or an end block.

8 -- End block
==========

format:
```
+--------------------------------+
| type: 8 |     length: 0        |
+--------------------------------+
```

an end block means the file can be closed, or the directory popped out of.

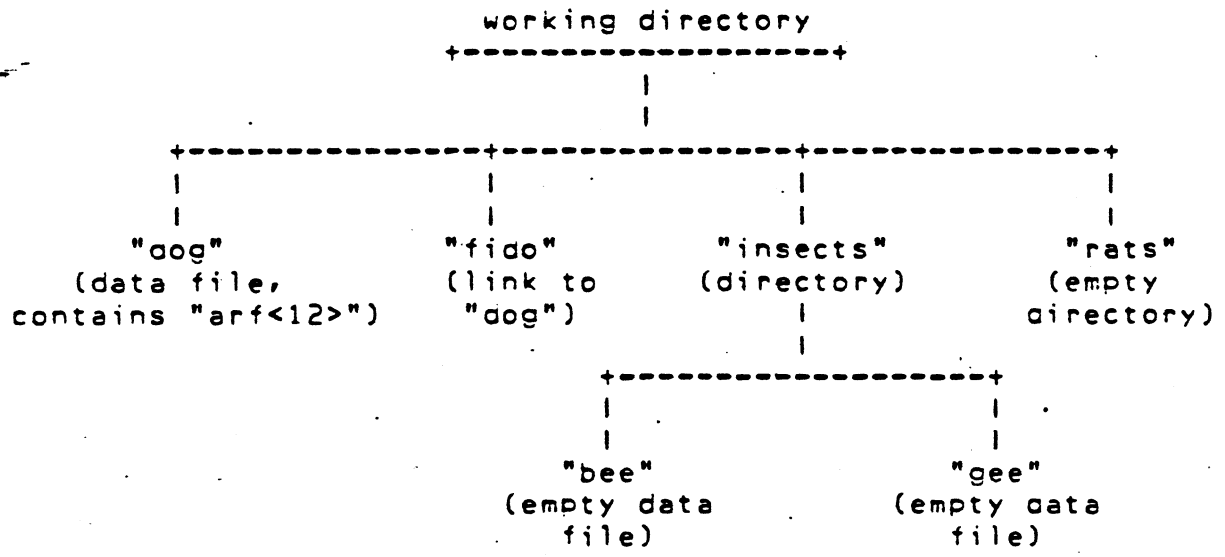
9 -- End of dump
==========

format:
```
+--------------------------------+
| type: 9 |     length: 0        |
+--------------------------------+
```

occurs only once, at the end of the dump file.

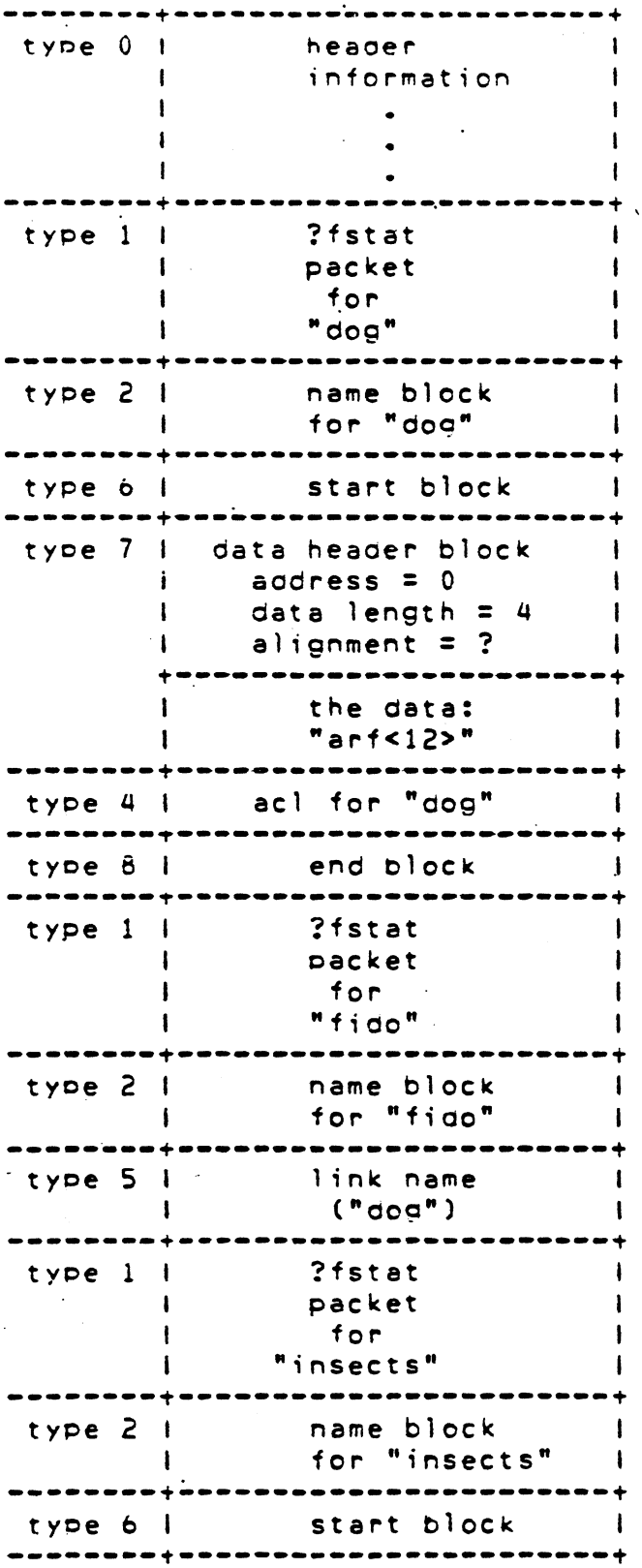
11.8.1 Picture of a sample dump file

Suppose we have the following directory tree:



suppose also that the file "gee" has a user data area (uda).

In the following diagram, the lines are not necessarily on word boundaries. Remember, the information 'blocks' are byte aligned, not word aligned.



type 1	?fstat packet for "bee"
type 2	name block for "bee"
type 6	start block
type 4	acl for "bee"
type 8	end block
type 1	?fstat packet for "gee"
type 2	name block for "gee"
type 6	start block
type 4	acl for "gee"
type 8	end block
type 3	uda for "gee"
type 4	acl for "insects"
type 8	end block
type 1	?fstat packet for "rats"
type 2	name block for "rats"
type 6	start block
type 4	acl for "rats"
type 8	end block
type 9	end of dump

11.8.2 Sample DEDIT of a dump file

The following is a real live sample of a CLI dump tape. The structure that was dumped is as follows:

```
DTR1
    FILE1
    FILE2
```

```
0 :000016       14.                     BKHDR (type 0)
1 :000017       15.                     Dump revision number
2 :000021       46.                     Second of tape creation
3 :000064       58.                     Minute of tape creation
4 :000000       22.                     Hour of tape creation
5 :000006       5.                      Day of tape creation
6 :000002       2.                      Month of tape creation
7 :000120       80.                     Year of tape creation
10 :002056      (000001 / 46.)          BKFST (type 1) / 46. bytes
11 :000012
12 :177777
13 :177777
14 :000007
15 :000000
16 :000000
17 :000001
20 :000003
21 :010503
22 :120560
23 :010503
24 :120623                              ?FSTAT packet (23. words long)
25 :010503
26 :120623
27 :000003
30 :000000
31 :011000
32 :000000
33 :020270
34 :000001
35 :000000
36 :000000
37 :000000
40 :004005      (000010 / 5)            BKNAM (type 2) / 5 bytes
41 :042111      DI
42 :051061      R1
43 :000030      <0>  (000110 /          BKBEG (type 6) / 0 bytes
44 :000004      000 )(000001 /          BKFST (type 1) / 46. bytes
45 :027000      46. )
46 :042377
47 :177777                              ?FSTAT packet
50 :177400
51 :000000
52 :000000
```

```
53 :000000          \
54 :000400           \
55 :001421            \
56 :041641             \
57 :072021              \
60 :041641               \
61 :111421                \
62 :041641                 \
63 :075000                  }          ?FSTAT packet (continue)
64 :001400                 /
65 :000000                /
66 :012400               /
67 :000040              /
70 :136000             /
71 :000000            /
72 :000000           /
73 :000000          /
74 :000010 +     000  { 000010 /        BKNAM (type 2) / 6 bytes
75 :003106 +     006  } F
76 :044514 +     IL
77 :042461 +     E1
100 :000030 +    <0>  { 000110 /        BKREG (type 6) / 0 bytes
101 :000034 +    0   }{ 000111 /        BKDAT (type 7) / 10. bytes
102 :005000 +    10. }
103 :000000 +          \
104 :000000 +           \
105 :000000 +            }              Data header information
106 :012400 +           /
107 :000400 +          /
110 :043111 +    FI   \
111 :046105 +    LE    \
112 :020061 +     1     \
113 :005106 +    <12>F   \
114 :044514 +    IL       \            Actual data
115 :042440 +    E        /
116 :030412 +    1<12>   /
117 :043111 +    FI     /
120 :046105 +    LE    /
121 :020061 +     1   /
122 :005020 +    <12> {000100 /         BKACL (type 4) / 10. bytes
123 :005044 +    10. } $
124 :042101 +    DA
125 :053105 +    VE
126 :000037 +
127 :025400 +
130 :001440 +    003  {001000 /         BKEND (type 10) / 0 bytes
131 :000004 +    000 }{000001 /         BKFST (type 1) / 46. bytes
132 :027000 +    46. }\
133 :042377 +         \
134 :177777 +          \                ?FSTAT packet
135 :177400 +          /
136 :000000 +         /
137 :000000 +        /
```

```
140 :000000 +        \
141 :000400 +         \
142 :001421 +          \
143 :041641 +           \
144 :076021 +            \
145 :041041 +             \
146 :111421 +              \
147 :041641 +               \
150 :077400 +                }          ?FSTAT packet (continued)
151 :001400 +               /
152 :000000 +              /
153 :012400 +             /
154 :000040 +            /
155 :137000 +           /
156 :000000 +          /
157 :000000 +         /
160 :000000 +        /
161 :000010 +   000  {000010 /          BKNAM (type 2) / 6 bytes
162 :003106 +   006} F
163 :044514 +   IL
164 :042462 +   E2
165 :000030 +   <0>  {000110 /          BKBEG (type 6) / 0 bytes
166 :000034 +   000} {000111 /          BKDAT (type 7) / 10. bytes
167 :005000 +   10.}\
170 :000000 +        \
171 :000000 +         \                 Data header
172 :000000 +         /
173 :012400 +        /
174 :000400 +       /
175 :043111 +   FI   \
176 :046105 +   LE    \
177 :020062 +    2     \
200 :005106 +   <12>F   \
201 :044514 +   IL       \            Actual data
202 :042440 +   E        /
203 :031012 +   2<12>    /
204 :043111 +   FI      /
205 :046105 +   LE     /
206 :020062 +    2    /
207 :005020 +   <12>}{000100 /          BKACL (type 4) / 10 bytes
210 :005044 +   10. } $
211 :042101 +   DA
212 :053105 +   VE
213 :000037 +
214 :025400 +
215 :001440 +   003  {001000 /          BKEND (type 10) / 0 bytes
216 :000020 +   000 }{000010 /          BKACL (type 4) / 10 bytes
217 :005044 +   10. } $
220 :042101 +   DA
221 :053105 +   VE
222 :000037 +
223 :025400 +
224 :001440 +   000  {001000 /          BKEND (type 10) / 0 bytes
225 :000044 +   000 }{001001 /          BKEDP (type 11) / 0 bytes
226 :000000 +   000 }
```

THIS PAGE ORIGINALLY WAS LEFT BLANK

CHAPTER 12 -- THE SYNC WORLD
(updated for AOS Rev. 3.11)

The synchronous world in AOS refers to the integrated synchronous communications driver. This chapter is a brief introduction to the design and operation of the synchronous system. It's scope has been deliberately limited as this is the first incorporation of the chapter in the manual. The chapter in future manuals will be an in-depth analysis into the design, operation, and system interface of this driver.

When designing an operating system control program the system must have a means by which it receives and transmits information. Without this input and output the system can do nothing. The sophistication of these input and output device drivers is left to the developers. You have already studied asynchronous I/O under the peripheral manager, mag tape systems and disk structures. Now, onward to communications.

12.1 The general function of a synchronous driver

In Asynchronous communications depressing a key at a terminal will send the character code to the host preceeded by a bit telling the host to wake up. The 7 or 8 bit (ASCII or EBCDIC) character is transmitted serially (bit-by-bit) and is followed by a bit or two telling the host it can go back to sleep again.

In synchronous communications large character groups or blocks are transferred in an organized manner, eliminating the large overhead of 3 bits of control for every 7 bit character. While Asynchronous communications line have a 33% control information overhead, synchronous overhead can be significantly less depending on the block size and the sophistication of the protocol, the means by which we communicate in an organized manner. IBM Binary Synchronous Communications (BSC) protocol and X.25, the industry standards, also provice other significant advantages. The most noticeable, of course, is they provide for error detection and correction. In the turbulent world of communications links it is not uncommon for bit patterns to be changed during transmission. Should an error occur the receiving station will simply ask for retransmission.

Before entering into a discussion on AOS you must understand the functions any synchronous controller must perform. The synchronous controller, like any other, must be able to transfer three types of information between it'self and the computer: control, data, and exception. This requires the driver to have the ability to distinguish the information as it comes over the link. This is not the case with most other controllers as the peripherals usually have separate wires to transmit these three types of information.

The means by which BSC protocol achieves a low overhead and error detection requires several other things of the system:

1) The system must be able to store large block of information before processing

2) The system must be able to determine if a character transliteration has occured. This is accomplished by a polynomial equation (using both vertical and horizontal check-sums) which both the transmitting and receiving stations compute. This redundancy check is then sent as the last two characters of a block by the transmitting station which is compared to the value the receiving station has computed. Appropriate acknowledgement is then made on the results of this block check. In our system the polynomials are computed by the Cyclic Redundancy Check (CRC) hardware on the Synchronous Line Multiplexer (SLM) boards.

The synchronous driver, as you can imagine, is different from any of the magnetic or asynchronous peripherals - there are no moving parts. The only hardware needed is a board and a modem at each end.

Communications protocol and modem control are beyond the scope of this chapter. If you would like to learn more about them the marketing training self-study manuals - An Introduction to Data Communications and Modem Control - are very informative.

12.2 The AOS synchronous driver

AOS unlike RDOS or AOS/VS contains an integrated synchronous driver. The advantage of integrating the driver into the system is that it is more efficient, and much easier to use. The major advantage achieved in a non-integrated driver is in system reliability. A separate driver might trap on an error while an operating system error will panic the system. Thus, the less functions the system has to perform the more reliable it will be. In addition, the more functions the system performs the longer Sysgen and boot time will take. The major reason AOS/VS has a separate driver is that the system could be announced earlier if the driver did not have to be included.

12.2.1 AOSGEN

The integration of a synchronous communications system begins at sysgen time when we specify the number of lines, the characteristics for each, whether we have a stand-alone or Data Comm Unit (DCU) assisted system, and the number of data channel pages we are going to allocate for the exclusive use of the synchronous system.

AOSGEN will produce a file - PID.IRMG.TMP - which consists only of a set of macro calls as defined by IRMG1 and IRMG2. This file is then assembled into a module which defines the unit tables, device control tables, and the peripheral tables. This file is then bound with the rest of the system files to produce the .sy file. The linker will only bind those device control tables and library three overlay modules that have been referenced to in the IRMG file. Addresses of undefined references are given a value of -1 by link.

12.2.2 SINIT activities

Like all other devices, the synchronous data structures must be initialized at boot time. Sinit runs through the unit tables examining and initializing each device. The relative page table, line table directory, and line table are all created during initialization.

12.3 Data Structures

12.3.1 Introduction

The primary data area used by the synchronous line drivers is a set of pages which are removed for the available memory pool during system initialization. The number of pages allocated is specified in answering the AOSGEN questions. A maximum of 28 pages may be allocated. All of the line drivers contend for the same set of pages. Memory is dynamically allocated from these pages to the line drivers as needed while the system is on-line.

The data structures for the synchronous lines are built both statically during system initialization and dynamically at run-time.

The RELATIVE PAGE TABLES, LINE TABLE DIRECTORIES, LINE TABLES, HOST TO SLAVE QUEUES(*), AND THE SLAVE TO HOST QUEUES(*) are the structures built during initialization, while all INPUT BUFFERS, OUTPUT BUFFERS, TRANSMISSION SHORT BUFFERS(*), TRANSMISSION FREE STACKS(*), TRANSMISSION QUEUES(*), RECEIVE INTERRUPT QUEUES(*), RECEIVE QUEUES(*), POLL LISTS(**), AND POLL SELECT ADDRESS TABLES(**) are built dynamically as needed.

The data structures marked with an asterick are for X.25 only while the poll select address tables are only used in a BSC multipoint environment. This release only covers the BSC point-to-point data structures.

12.3.2 The relative page table

The relative page table is of the form:

word	contents
0	100040
1	unused
2	unused
3	unused
4	RPT P#
5	page/add
.	.
.	.
.	.
40	.
41	DCT ad
42	"
43	"
44	1730

The maximum number of pages allocatable is 28. This is because the DCU uses four pages to map it's own memory.
45=32. + 5.

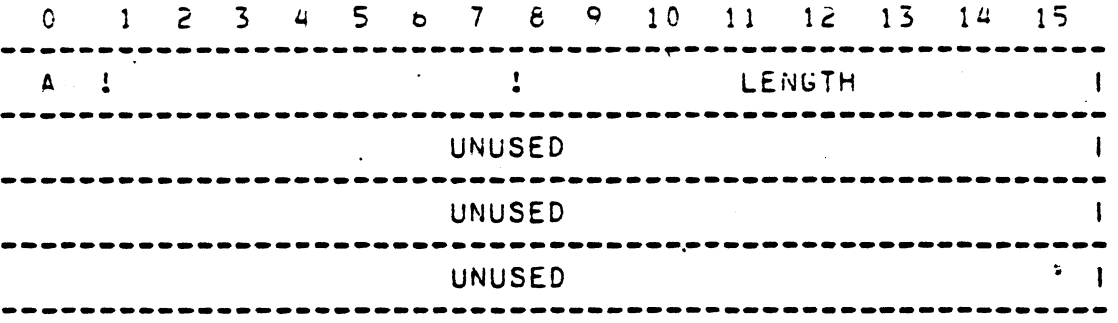
The relative page table is used to keep track of the physical page addresses of the synchronous line data area. There are 28 entries in this table. Each entry contains a 10 bit physical page address (right justified). If there are more entries in the table than pages allocated the unused entries are marked invalid.

All of the data structures located in the synchronous line data area are addressed by an 8 bit relative page address concatenated with an 8 bit offset within the page. The relative page address is used as an index into the relative page table. Currently this index must be in the range 0 to 27 because the DCU has only the 8 bit offset value represents the actual offset divided by 4. Hence the relative page table provides the mapping from the internal virtual address to the physical page address.

The relative page table is constructed during system initialization. It is located in the first page allocated to the synchronous line data area. In order to find the relative page table the address of the first page allocated, which contains the relative page table, is stored in the device control tables (DCT's) for all of the DCU's and/or stand-alone SLM's in the system.

12.3.3 Memory Space Headers

Memory space headers are used by the synchronous line data area memory allocation/detail/deallocation routines to keep track of what areas of the data area are available and what areas are used. Each memory space header consists of 4 words in the following format:



Where,

A specifies whether the data area is available or not. If A is 0, the space is available, if A is 1, the space is used.

LENGTH specifies the number of words in the block of memory described by the header. This does not include the bytes making up the header.

When the pages are allocated to the synchronous line data area during system initialization, each page is initialized with one header at the beginning of the page. The available flag is set to 0, indicating the block contains the entire page. As blocks of memory are required headers are created, modified, and deleted. There will always be one header at the front of each page.

12.3.4 Input/Output buffers

There is a set of input buffers and a set of output buffers for each synchronous line. These buffers are dynamically allocated from the synchronous line data area while the system is on line. Input buffers are used to contain the data received from the synchronous line. Output buffers are used to contain the data to be transmitted onto the synchronous line.

The format of the input and output buffers are the same. All buffers are addressed by a virtual address. This address is an 8 bit field representing the relative page number for the page in which the buffer resides concatenated with an 8 bit field representing the offset within the page divided by four of the start of the buffer. Each buffer has six word header followed by space for

data. The header is used for coordinating operations between the line driver and the DCU. It has the following format:

word 0: link
word 1: buffer identifier
word 2: buffer size
word 3: data byte count
word 4: frame status
word 5: pointer to frame

The link entry is used for threading buffers onto queues. Link addresses are virtual as described above.

The buffer identifier entry is used to associate a user specified identifier for the buffer. This identifier is specified by the applications line driver when it issues a send continue call. When control is returned to the caller the identifier of the buffer which has been most recently transmitted is returned to the caller.

The buffer size entry is used to contain the size of the buffer itself in bytes.

The data byte count entry is used to specify the number of data bytes contained in the buffer. It does not include the header.

The frame status entry is used to maintain information regarding the type of frame the buffer contains (i.e. I frame, RR frame, UA frame, etc.). It is also used to specify whether the buffer was sent with line turn around (hdx only), whether a buffer overrun occurred when the DCU filled the buffer, and whether the DCU has finished with the buffer.

When buffers are allocated the size of the data area is defined. This size is specified by the application line handler when the line for which they will be used is enabled. All buffers for a given line will have the same size. The maximum size allowed is 1014 bytes (plus 6 for the header) because the buffer must fit on one page. Buffers are not split over physical page boundaries.

CHAPTER 13 - MISCELLANEOUS TOPICS
(updated for AOS Rev. 3.11)

13.1 Networking

The following section documents how AOS handles networking. It does not document the actual RMA or x25 code, but rather the code internal to AOS.

13.1.1 Some key networking locations

LHID	(SZERO)	local host id
LHNAME	(STABLE)	local host name
NETCCB	(STABLE)	CCB for :NET
HIFCCB	(STABLE)	CCB for :PROC:HIF
RPID	(STABLE)	PID of RMA process
RPHI	(STABLE)	HI portion of IPC port for RMA
RPLO	(STABLE)	LO portion of IPC port for RMA

13.1.2 The Host Information File (HIF)

The HIF resides in :PROC, and is created at SINIT time (see network initialization below). The file is indexed by host ID and is used to retrieve the host name associated with that host ID. Each HIF entry is 128 words long. The indexing algorithm is as follows:

Block # in file = (HID-1)/2

Offset in block = (0 if HID is odd) or (200 if HID is even)

Free entries are indicated by a 0 in the first byte.

13.1.3 Network Initialization

The network initialization code is called from SINIT at system initialization time. The code is in the overlay NETI1, and the entry point is NETI.

1. Create :NET (if create returns an error, ignore it, we assume the directory already exists.)

2. Allocate a system CCB for :NET and store its address in NETCCB.

3. Create :PROC:HIF (if we get an error, abort.)

4. Get a system CCB for :PROC:HIF and store its address in HIFCCB.

5. Scan the :NET directory looking for all files of type ?FREMA type. For each file of ?FREMA type, create an entry in :PROC:HIF by calling CHIFE (see below).

6. Return back to the SINIT code.

13.1.4 AOS <--> RMA interface

The interface between AOS and the RMA process is located in the overlay SRI. Upon AOS's detection of a network request, the individual code paths chain to SRI to complete the system call.

1. Call the general setup routine for network calls. This will:

   a. If LHID = -1 (no local host), set all RMA indicators to -1 to indicate that RMA is not around, and return the 'Network not available' error.

   b. If RMA is not around (RPID = 0), try and find it by doing a lookup on the IPC entry for RMA. If found, verify that it is indeed a IPC entry, and store the PID associated with it in RPID. Store the RMA global port number in RPHI/RPLO

   c. Clear the IPC packet and the internal buffer used to communicate with RMA

   d. Initialize some of the constants in the IPC header (i.e. length of message ...)

2. Set up the remainder of the IPC message packet. This includes the

   a. HID (?RHID)
   b. PID (with the ghost indicator bit) (?RTCB)
   c. TCB address (?RTCB)
   d. AC0 (?RAC0)
   e. AC1 (?RAC1)
   f. AC2 (?RAC2)
   g. System call word (?RSYS)

3. Put superuser/superprocess status into leftover bits in ?RHID.

4. Connect the caller (customer) to RMA.

5. Mark the user's TCB as awaiting wake up by ?IWKUP

6. Send the IPC message to RMA

7. Return.

15.1.5 Some key internal networking routines

(in NETI1) -- Keyboard interrupt handling

. Get the interrupt task address (411 in UST) of the PTBL associated with the current CB.

. Validate the address 423=< addr =< 623

. Unpend the task.

. Put the interrupt character in ?TAC0 of the interrupt task

. Reset the sched action bit in the PTBL

INTR (in NETI1) -- Force a simulated interrupt on a process.

1. Get the interrupt character from AC0 of the calling TCB.
2. Get the victim PID from AC1 of calling TCB.
3. Verify PID [PID =< PIDLN, PID exists in PIDBT]
4. Get the PID of the person making the call, and form a PID couplet Victim PID / Current PID)
5. Test for connection (if none, abort)
6. Map in target PTBL and call IWKUP.
7. If ?KIOFF is enabled, we are done.
8. If ?KWAIT is up then:
   a. Set interrupt received.
   b. Set daemon start.
   c. If the target (victim) process is blocked, unblock it.
   d. We are done.
9. Validate the interrupt character (^A =< char. =< ^E). If not, we are done.
10. If ^C or ^D we are done.
11. If not ^A, then check the OP interrupt disable bit and if it is not set, the set the delayed interrupt bit (BPFDI), we are done.
12. Set the interrupt received bit.

13. If ^A, set the interrupt bit (BPFIN)
    If ^B, set the term bit (BPFTM)
    If ^E, set the term bit and breakfile request bit (BPFBR)

14. If the process is blocked, unblock it.

15. Return

CHOST - Check host. Takes a byte pointer to a process name and determines if it refers to a remote or local process

1. Allocated 80 bytes for temporary host name

2. Call RDM70 to map in the user pages that contain the process name

3. Validate the process name string. Must be in the format
   <Host name>:<username>:<process name><0>

4. Look up the host name in :NET. If the host name is not in :NET (or it is but not of type ?FREMA), then take the non-network return.

5. Get the HID from the FIB (offset 10) and pass it back to the caller.

6. Put 367 [Network reference error] into the CB (offset CERWD), and take the error (network) return.

CHIFE (in NET1) -- create a HIF file entry
DHIFE (in NET1) -- delete a HIF file entry

1. Set mode flag (0 if create, 1 if delete)

2. Validate HID (1=< HID =< 127)

3. Lock the HIF CCB.

4. Compute the block that the host name is/will be in.

5. Read in the block

6. Calculate the inblock offset (if odd then 0, else 200 for even)

7. If this is a call to delete (mode = 1):

   a. If the entry is already free, return error.
   b. Zero the 1st byte of the entry to indicate it is free.
   c. If the entry we are deleting is the local host, move -1 into LHID and zero the first byte of the local host name string.

8. If this is a call to create (mode = 0):

   a. If HID is already defined, return error.
   b. If we are trying to create the local HID: if one already exists, error, else initialize the LHID and local hostname
   c. Move the host name string into the appropriate block in HIF.

9. Flush the modified block back to the HIF.

10. Unlock the HIF CCB, and take the good return.

HHNAME -- ?HNAME system call

Upon entry, AC1 contains an entry code:

1 = Given HID, return Hostname, and length of Hostname
2 = Given hostname, return the HID.
3 = Return LHID and local host name

Code 1:

1. Validate HID (in range?)

2. Lock the CCB for HIF

3. Get the appropriate block of the HIF for the given HID.

4. If the first byte of the appropriate offset is 0, then return error 364 [HID not defined].

5. Move the string in the HIF into the user space, and put the string's length into the users AC1.

Code 2:

1. Lookup the host name in :NET. If not there (or there but not a ?FREMA type file), return error 'Host does not exist'.

2. Get the HID from the FIB associated with the :NET entry (offset 10), and put it in the users AC1.

Code 3:

1. If the LHID is defined, put it in AC1, else return an error.

2. If the user requested the hostname, move the string into the user's space, and put the string's length into AC2.

13.2 AP support

AOS supports primitive use of the Array Processor hardware on the AP/130 and on the S/250. Because the firmware offers no lock bit, any user with I/O privileges can use the hardware. AOS provides a convention to prevent multiple users from manipulating the AP simultaneously, but the users must follow the rules.

13.2.1 AP page zero locations

There are currently three locations in page zero used in for AP support:

APLOD: -1 if the AP is not loaded, 0 if it is.
APSIZ: contains the amount of memory required by the AP or -1 if the machine does not support the AP.
APPID: contains the PID number of the current user of the AP

13.2.2 AP during system initialization

AOS reserves APSIZ pages for the AP during system initialization. These are always the last APSIZ pages of physical memory. Note that if we also need a physical page for the diskworld's ZMAP, it will come below the AP pages ([APSIZ + 1] from the top)

13.2.3 AP user interfaces

Revision 3.11 introduced four new system calls that allow users to access the AP. These are documented below.

?APINIT

The calling process becomes the sole owner of the AP. If the AP exists (APSIZ<>-1) and the APPID is either the calling process or 0, then set APPID equal to the calling process, ISZ the 'Do not swap counter' and take the good return. If APSIZ=-1 then return 'AP does not exist'. If APPID does not equal 0 or the callers PID, then return 'AP busy'.

?APREL

The calling process gives up sole ownership of the AP. If the AP is defined, and the calling process owns it, then clear APPID. AOS then releases the pages associated with the AP in the users shared area. Finally, the 'Do not swap counter' is decremented, and the good return is taken. If APSIZ=-1, then return 'AP does not exist'. If APPID<> current PID, then return 'AP busy'.

?APMAP

Map the array processor pages into the user's shared area. First, check to make sure the AP exists and is loaded (APSIZ<>-1 and APLOD<>-1). Next, validate the request (# of pages<APSIZ, starting page < APSIZ, request is paged aligned and in shared area). Then, step through the requested pages making sure that they all lie in the users shared area, and that the pages are AP pages. AP requests wrap around, implying that if you request 3 pages, starting at AP page 3, you will get pages 3,4,1 mapped in your shared area.

?APOK

Set or retrieve status of AP microcode. If the AP exists and the request is for status, return the value of APLOD in AC2 and the value of APPID in AC1. If the AP exists, and the request is to 'turn' the AP on or off, and the user has I/O privileges, then set APLOD to the appropriate value.

13.3 SYSBOOT

SYSBOOT is a utility program that is used to bootstrap AOS into memory. SYSBOOT initially lives in physical pages 0 - 30. It then copies itself up to the top of memory, and reads the system in below itself. Finally, it executes a SVC instruction, and since location 2 now contains the value of SINIT, the AOS system will start initializing. Below is a detailed summary of the SYSBOOT code path.

1. Put the device code of the desired disk in the switches and hit PROGRAM LOAD. This causes the first block of DSKBT (block 0) to be read in and executed. This block contains a disk driver and enough smarts to read in the second half of DSKBT (block 1).

2. DSKBT then reads up the DIB of the disk (block 3) and fetches the address (IBSBH IBSBL) and size (IBSSB) of the system bootstrap area. It then reads in the system bootstrap and transfers control to it.

3. SYSBOOT is loaded into physical pages 0-30. SYSBOOT must be less than 28 pages long, since it needs up to 2 pages to hold bad block tables (8*256.= 2048. = 2 pages), one page to hold index blocks, and one page to use as a communications area to SINIT.

4. The first thing SYSBOOT does is call "MINIT" which sets up the data channel map to be the identity map and returns "HPAGE", which is equal to 127, the number of the highest page we will be working with.

5. "MINIT":
   a) Checks the size of main memory by starting at the top and setting the page number in the first 8 words of pages 64-127 since there can be up to 8-way interleaving memory, and removing one board might remove just every 8th word from memory. Or course if every 8th word is missing, we might have problems even executing "MINIT".

   b) Reads the page numbers in ascending order and stops if it finds a discrepancy (insufficient memory).

   c) Initializes and enables the data channel map to be the identity map for the first 32kw.

Logical	Physical
0	0
1	1
2	2
.	.
.	.
.	.
28	28
29	29
30	30
31	HPAGE

6. SYSBOOT's main routine now executes:

   a) A initialization routine loops asking for disk unit names. If the user makes a mistake, we simply reject that disk and go on asking. If, however there is an inconsistency in a device information block (DIB), we stop.

   b) Reads DIB of first PU and checks if FIXUP needs to be run.

   c) Sets up pointer to funny FIB and moves it in.

   d) Builds a table in physical page 31. for the system that provides the disk overlay area addr. and describes each PU in the LDU. Each PU element consists of 4 words containing device name and device code. For example, a Zebra entry might look like:

```
76000: disk addr. of overlay area. HI
76001:   "    "   "       "     "   LO
76002:  D    P
76003:  F    1
76004:  0   <0>
76005:     <67>. (device code)
760XX: =0, table terminator
```

   e) Gets system (or other program) name to boot and sets up intra-directory pointer to the file.

f) Finishes table for SINIT.

76043: disk addr. of block that contains system FIB. HI
76044: " " " " " " " " LO
76045: offset block within the above block that contains FIB.

g) Reads in systems users status table (UST).

h) Copies overlays to space reserved for it.

i) Calls "WARP" which gets SYSBOOT out of low memory. This routine supervises the jump to hyperspace.

1. Figures out base of new home. HPAGE-31

2. Next map enabled will be user A.

3. BLM's itself up near HPAGE.

4. Moves a routine to HPAGE.5 (77000) and jumps to it which:

a) Zeros first 31kw of memory.

b) Loads and enables user map A.

c) Returns to SYSBOOT (user map A).

j) SYSBOOT then maps data channel slot 31 to physical page HPAGE, where it stays for the rest of it's life. It is here that the index blocks needed to read in the system are hidden. At this point SYSBOOT looks like this:

Logical	Physical
0	HPAGE-31
1	HPAGE-30
2	HPAGE-29
.	.
.	.
.	.
28	HPAGE-3
29	HPAGE-2
30	HPAGE-1
31	HPAGE

k) The data channel map is swept across memory to read in as much of the system as will fit. This is done 31 pages at a time (i.e. data channel slots 0-30 are used), then the data channel is set up to point 31 pages higher than it was.

l) When the system is read in SYSBOOT transfers control to either "GATE" or "DIE".

m) When SYSBOOT asked you which system to load, if you preceded your response with a backslash (\), then SYSBOOT loads your system but will not execute it. "DIE" is called when we don't want to transfer control to the system. It BLM's HALTs thru SYSBOOT's address space and halts.

n) "GATE" transfers control to the system which has been loaded into physical pages 0-30. Thru fantastic coincidence and luck (and maybe actually foresight), we use SVC to transfer control to the system. The SVC instruction disables the map, doesn't push a control block onto the stack and does a jump indirect to location 2, which contains the address of SINIT. We're off !!!

```
02                ; COPYRIGHT (C) DATA GENERAL CORPORATION 1977,1978,1979,1980
03                ; ALL RIGHTS RESERVED.
04                ; LICENSED MATERIAL-PROPERTY OF DATA GENERAL CORPORATION
17                ;====================================;
18                ; AOS REVISION  3 SYSTEM PARAMETERS ;
19                ;====================================;
20
21
22
23                ; SYSTEM SUBROUTINE LINKAGE PARAMETERS
24
25          127710 .DUSR    RTRN=   RTN       ;CALL TO RESTORE STATE
26          000000 .DUSR    ORTN=   0         ;RETURN LOCATION
27          177774 .DUSR    OAC0=   -4        ;CALLER'S AC0
28          177775 .DUSR    OAC1=   -3        ;CALLER'S AC1
29          177776 .DUSR    OAC2=   -2        ;CALLER'S AC2
30          177777 .DUSR    OSP=    -1        ;CALLER'S CSP
31          177777 .DUSR    OFP=    -1        ;CALLER'S FP
32          000001 .DUSR    TMP=    1         ;CALLEE'S FIRST TEMPORARY
33          177767 .DUSR    ROAC4=  -11       ;CALLER'S LAST TEMP (AFTER ROVCL)
34          177773 .DUSR    VOAC4=  -5        ;CALLER'S LAST TEMP (NOT OVLY CALL)
35          177771 .DUSR    OAC4=   -7        ;CALLER'S LAST TEMP (ON OVLY CALL)
36          177770 .DUSR    OAC5=   -10       ;CALLER'S NEXT TO LAST TEMP (ON OVLY)
37
38                ; SYSTEM CONTROL PARAMETERS
39
            000400 .DUSR    SLGTH=  256.      ;SYSTEM STACK LENGTH
            000100 .DUSR    SSLGT=  100       ;INTERRUPT STACK LENGTH
            000144 .DUSR    SCDCG=  100.      ;10 MILL DISK CHARGE
43          000372 .DUSR    SCMCG=  250.      ;25 MILL MAG TAPE CHARGE
44          000012 .DUSR    SCSCG=  10.       ;1 MILL
45          000005 .DUSR    SCNCB=  5.        ;NUMBER OF CONTROL BLKS
46          000006 .DUSR    SCOVM=  6         ;MIN NUMBER OF OVERLAY AREAS
47          000014 .DUSR    SCBFM=  SCNCB*2+2       ;MIN NUMBER OF BUFFERS
48                                                  ; (CB * 2 TO PREVENT DEADLOCKS)
49          000144 .DUSR    SCNSO=  144       ;MAX NUMBER OF SYSTEM OVERLAYS
50          000002 .DUSR    SCNRO=  2.        ;# RES SYS OVLYS FOR SYSTEM INIT
51          003720 .DUSR    SCLMX=  2000.     ;# SHARED PAGES ON LRU
52                                            ; EFFECTIVELY UNLIMITED UNTIL CODE IS
53                                            ; REMOVED
54
55
56          000002 .DUSR    SCFRO=  2         ;NUMBER OF FIRST ROVCL OVERLAY
57                                            ;THIS MUST FOLLOW INIT OVERLAYS
58          000043 .DUSR    SCNOR=  35.       ;NUMBER OF RESIDENT OVERLAYS
59                                            ;FOR USE BY ROVCL
60          000004 .DUSR    SCSLN=  4         ;NO OF SYNC LINE RES OVLYS
```

```
0002  PAR
01          000400 .DUSR    [illegible]=  256.      ;MAXIMUM LENGTH FOR PORT TABLES
02          000015 .DUSR    SCRTY=  15      ;MAX NUMBER OF DEV RETRYS
03          000003 .DUSR    SCTMO=  3       ;UNIT TIMEOUT VALUE (SECONDS)
04          000037 .DUSR    SCFHS= 31.      ;FCB HASH FRAME SIZE
05
06
07                  ; SYSTEM PAGE ZERO LOCATIONS
08                  ;       LOCATION 1 IS THE INTERRUPT HANDLER
09                  ;       LOCATION 2 IS THE SYSTEM ENTRY POINT
10                  ;       LOCATION 3 IS THE PROTECTION FAULT ROUTINE POINTER
11
12          000004 .DUSR    ISP=    4       ;INITIAL INTERRUPT STACK POINTER
13          000005 .DUSR    CMSK=   5       ;CURRENT INTERRUPT PRIORITY MASK
14          000006 .DUSR    ISL=    6       ;INTERRUPT STACK LIMIT
15          000007 .DUSR    ISO=    7       ;INTERRUPT STACK OVERFLOW ADDRESS
16          000010 .DUSR    FBK=    10      ;FAULT BLOCK POINTER
17
18                  ; THE FOLLOWING THREE WORDS ARE RESERVED BY THE HARDWARE
19                  ; FOR THE E-500, AND DEFINED IN BOTH E-200 & E-500.AOS
20                  ; FOR COMPATABILITY.
21
22          000011 .DUSR    PFHAN=  11      ;ADDRESS OF PAGE FAULT HANDLER
23          000012 .DUSR    BPHAN=  12      ;ADDRESS OF HARDWARE BREAKPOINT HANDLER
24          000013 .DUSR    VBPTR=  13      ;ADDRESS OF VALIDITY BITS
25
26          000365 .DUSR    CC=     365     ;CURRENT CONTROL BLOCK POINTER
27          000366 .DUSR    CRSEG=  366     ;CURRENT OVERLAY POINTER
28
29                  ;.DUSR  SP=     40      ;HARDWARE STACK POINTER
30          000041 .DUSR    CSP=    41      ;FRAME POINTER
31          000042 .DUSR    CSL=    42      ;STACK LIMIT
32          000043 .DUSR    CSO=    43      ;STACK OVERFLOW ROUTINE ADDRESS
33
34                  ;.DUSR  FP=     CSP     ;ANOTHER NAME
```

```
10003  PARS
01
02              ; BUFFER ENTRY
03              ; BE WARNED THAT A CHANGE TO THE BUFFER HDR MAY IMPLY A CHANGE
04              ; TO THE CORE BLK HDR....

        000000 .DUSR    BQADR=0         ;DATA ADDR (MUST BE OFFSET 0)
        000001 .DUSR    BQST=1          ;BUFFER STATUS
08      000002 .DUSR    BQQLK=2         ;DRIVER ENQUEUE LINK
09      000003 .DUSR    BQNBK=3         ;NUMBER OF BLKS (IF NOT SYSTEM BUFFER)
10      000004 .DUSR    BQMAP=4         ;IF 180, BQMAP CONTAINS PHYSICAL BLK #
11                                      ;IF 080, BQMAP CONTAINS PROCESS TABLE ADD
12                                      ;(MAP IS AT OFFSET PAMAP FROM THIS ADDRES
13                                      ;(CHANGED SO CCB'S CAN BE MAPPED AT I/O T
14      000005 .DUSR    BQUPD=5         ;UNPEND ROUTINE ADDRESS
15      000006 .DUSR    BQTCB=6         ;TCB ADDRESS
16      000007 .DUSR    BQLAH=7         ;LOGICAL ADDR (HIGH)
17      000010 .DUSR    BQLAL=10                ;LOGICAL ADDR (LOW)
18
19              ; FOR CONTINUITY WITH CORE HEADER, BQ LA PAIR MUST BE LAST
20              ; ENTRY IN COMMON HEADER
21
22      000011 .DUSR    BQDBN=11        ;DATA BLK NUMBER
23      000012 .DUSR    BQFFL=12        ;BUFFER LIST FORWARD LINK
24      000013 .DUSR    BQFBL=13        ;BUFFER LIST BACKWARD LINK
25      000014 .DUSR    BQUSC=14        ;USE COUNT
26      000015 .DUSR    BQBFL=15        ;LRU CHAIN FRONT LINK
27      000016 .DUSR    BQBBL=16        ;LRU CHAIN BACK LINK
28
29              ; SOME REDEFINITIONS
30

        000002 .DUSR    BQERR=  BQQLK   ;ERROR CODE (ON CALL TO POST PROCESSOR)
        000003 .DUSR    BQFCB=  BQNBK   ;V(FCB) ADDRESS (IF A SYSTEM BUFFER)
        000011 .DUSR    BQBMA=  BQDBN   ;POINTER TO SPECIAL REQUEST TYPE
34                                      ;PACKET (HAS BIT MAP AND WORK AREA)
35      000015 .DUSR    BQLCB=  BQBFL   ;LCB ADDRESS WHEN ENQUED
36      000011 .DUSR    BQSBP=  BQDBN   ;POINTER TO STATUS BUFFER FOR PIO
37
38              ; MAG TAPE REDEFINITIONS
39
40      000007 .DUSR    BQFIL=  BQLAH   ;FILE NUMBER
41      000010 .DUSR    BQBLK=  BQLAL   ;BLOCK NUMBER
42      000011 .DUSR    BQRBC=  BQDBN   ;RUNNING BYTE COUNT
43      000013 .DUSR    BQPRI=  BQFBL   ;PRIORITY
44      000014 .DUSR    BQERC=  BQUSC   ;ERROR COUNT
45      000015 .DUSR    BQCCB=  BQBFL   ;CCB ADDRESS
46      000016 .DUSR    BQUDB=  BQBBL   ;UDB ADDRESS
47      000017 .DUSR    BQWDC=  17      ;WORD COUNT
48
49              ; MCA REDEFINITIONS
50
51      000007 .DUSR    BQLNK=  BQLAH   ;LINK NUMBER (LEFT BYTE)
52      000007 .DUSR    BQSTT=  BQLAH   ;STATE OF I/O (RIGHT BYTE)
53      000010 .DUSR    BQRTY=  BQLAL   ;TIMEOUT RETRY COUNT
54      000012 .DUSR    BQOWC=  BQFFL   ;ORIGINAL WORD COUNT
55      000014 .DUSR    BQVCB=  BQUSC   ;VIRTUAL CCB ADDRESS
56      000016 .DUSR    BQDCT=  BQBBL   ;DCT ADDRESS
                                        ; 180 --> RECVR DCT
                                        ; 080 --> XMTR DCT
60              ; LPB REDEFINITIONS
```

```
01
02          000007 .DUSR    BQTMP=  BQLAH   ;TEMPORARY
03          000012 .DUSR    BQORC=  BQFFL   ;ORIGINAL BYTE COUNT
04          000014 .DUSR    BQBYT=  BQUSC   ;BYTE FLAG
05          000017 .DUSR    BQQBC=  BQWDC   ;REQUESTED BYTE COUNT
06
07                  ; BUFFER STATUS BIT POSITIONS
08                  ; HIGH ORDER 8  BITS ARE COMMON TO CORE BLK HDRS
09
10          000000 .DUSR    BSMOD=  0.      ; MODIFIED
11          000001 .DUSR    BSIOP=  1.      ; I/O IN PROGRESS
12          000002 .DUSR    BSROD=  2.      ; READ ONLY DATA
13          000003 .DUSR    BSERR=  3.      ; ERROR
14          000004 .DUSR    BSFLS=  4.      ; FLUSH BUFFER ON LAST RELEASE
15          000005 .DUSR    BSXIO=  5.      ; EXTENDED I/O (> 32K)
16          000006 .DUSR    BSNAB=  6.      ; NOT A SYSTEM BUFFER
17          000007 .DUSR    BSPIO=  7.      ; PHYSICAL I/O REQUEST
18          000012 .DUSR    BSCHD=  10.     ; CORE BLK HDR
19          000013 .DUSR    BSUPR=  11.     ; UNPEND UPON COMPLETION OF I/O (CORE HDR
20          000014 .DUSR    BSSRT=  12.     ; SPECIAL REQUEST TYPE(DISK)
21          000015 .DUSR    BSEOV=  13.     ; ENABLE VFU LOAD / OVERIDE LEOT LOGIC
22          000016 .DUSR    BSRT1=  14.     ; REQUEST TYPE BIT 1
23          000017 .DUSR    BSRT2=  15.     ;    "     "    "  2
24
25                  ;BYTE POINTER FOR RIGHT BYTE OF BQERC WORD.
26
27          000031 .DUSR    BHRBQ=  BQERC*2.+1      ;ADDR. OF RIGHT BYTE OF BQERC.
28
29                  ;DENSITY MODE BIT MASK.
30
31          014000 .DUSR    DBMSK=  1B3+1B4
32
33                  ;BQERC WORD BIT MEANINGS.  THEY HANDLE DENSITY MANAGEMENT FOR
34
35          000300 .DUSR    BQDCB=  BQERC*16.+0.    ;DENSITY CHANGE BIT.
36          000301 .DUSR    BQHCD=  BQERC*16.+1.    ;HAVE CHANGED DENSITY BIT.
37                                                  ;BITS 2-4 ALLOCATED FOR DENSITY M
38          000305 .DUSR    BQLCK=  BQERC*16.+5.    ;FILE/TAPE DENSITY LOCK.
39          000306 .DUSR    BQDEL=  BQERC*16.+6.    ;FILE/TAPE DENSITY ERROR LOCK.
40
41                  ; STATUS BLOCK FOR PHYSICAL I/O (PT'D TO BY BQSBP IN BH)
42
43          000000 .DUSR    PRBB=   0       ;RELATIVE BLK IN XFER OR BAD BLOCK
44          000001 .DUSR    PCS1=   PRBB+1  ;CONTROLLER STATUS WORD 1
45          000002 .DUSR    PCS2=   PCS1+1
46          000003 .DUSR    PCS3=   PCS2+1
47          000004 .DUSR    PCS4=   PCS3+1
48          000005 .DUSR    PCS5=   PCS4+1
49          000006 .DUSR    PCS6=   PCS5+1
50          000007 .DUSR    PCS7=   PCS6+1
51          000010 .DUSR    PCS8=   PCS7+1
52
53          000011 .DUSR    PIBLT=  PCS8+1  ;SIZE OF STATUS BLK
```

```
10005  PARS
01
02
03              ;  BUFFER STATUS MASKS
04
      100000 .DUSR    QTMOD=   1B(BSMOD)        ; BUFFER MODIFIED (MUST BE BIT 0)
      040000 .DUSR    QTIOP=   1B(BSIOP)        ; I/O IN PROGRESS
      020000 .DUSR    QTROD=   1B(BSROD)        ; READ ONLY DATA
08    010000 .DUSR    QTER=    1B(BSERR)        ; ERROR DETECTED
09    004000 .DUSR    QTFLS=   1B(BSFLS)        ; FLUSH BUF ON LAST RELEASE
10    002000 .DUSR    QTXIO=   1B(BSXIO)        ; EXTENDED I/O
11    001000 .DUSR    QTNAB=   1B(BSNAB)        ; NOT A SYSTEM BUFFER
12    000400 .DUSR    QTPIO=   1B(BSPIO)        ; PHYSICAL IO
13    000040 .DUSR    QTCHD=   1B(BSCHD)        ; CORE HEADER
14    000020 .DUSR    QTUPR=   1B(BSUPR)        ; UNPEND UPON COMPLETION (FOR COR
15    000010 .DUSR    QTSRQ=   1B(BSSRT)        ; SPECIAL REQUEST TYPE FLAG
16    000004 .DUSR    QTEOV=   1B(BSEOV)        ; ENABLE VFU / OVER RIDE LEOT
17    000002 .DUSR    QTRT1=   1B(BSRT1)        ; REQUEST TYPE BIT 1
18    000001 .DUSR    QTRT2=   1B(BSRT2)        ;    "     "    "  2
19
20
21              ;       REQUEST TYPES
22
23    000000 .DUSR    QTIO=    0                ; I/O (QTMOD=1 => WRITE)
24    000001 .DUSR    QTSTS=   1                ; GET STATUS
25    000002 .DUSR    QTREC=   2                ; RECALIBRATE OR REWIND
26
27
28              ;  BUFFER STATUS BIT POINTERS
29
30    000020 .DUSR    BQTMD=   BQST*16.+BSMOD   ; BUFFER MODIFIED
      000021 .DUSR    BQTIO=   BQST*16.+BSIOP   ; I/O IN PROGRESS
      000022 .DUSR    BQTRO=   BQST*16.+BSROD   ; READ ONLY
      000023 .DUSR    BQTER=   BQST*16.+BSERR   ; ERROR DETECTED
34    000024 .DUSR    BQTFL=   BQST*16.+BSFLS   ; FLUSH ON LAST REL
35    000025 .DUSR    BQXIO=   BQST*16.+BSXIO   ; EXTENDED I/O
36    000026 .DUSR    BQNAB=   BQST*16.+BSNAB   ; NOT A BUFFER
37    000027 .DUSR    BQPIO=   BQST*16.+BSPIO   ; PHYSICAL IO
38    000032 .DUSR    BQCHD=   BQST*16.+BSCHD   ; CORE HEADER
39    000033 .DUSR    BQTUO=   BQST*16.+BSUPR   ; UNPEND UPON COMPLETION
40    000034 .DUSR    BQSRQ=   BQST*16.+BSSRT   ; SPECIAL REQUEST TYPE
41    000035 .DUSR    BQEOV=   BQST*16.+BSEOV   ; FVU / LEOT
42
43
44    000020 .DUSR    BHDLN=   16.              ; BUFFER HEADER LENGTH
45    000011 .DUSR    CHDLN=   BQLAL+1          ; STK HDR LENGTH
```

```
:0006  PARS
01              ;.SYSTEM EVENT LOGGING Q PARAMETERS AND VALUES.
02              ;
03              ; THE Q AREA CAN BE MADE LARGER IN QUARTER PAGE CHUNKS UP TO 1 P
04              ; CURRENTLY, WE ALLOCATE 1 QUARTER PAGE (10/26/79).
05              ; TO DO THIS, DO THE FOLLOWING:
06              ;       1. CHANGE THE Q CONTROL VALUES AND GENERATE A NEW PS R
07              ;       2. CHANGE SINIT(WHERE THE SPACE IS ALLOCATED).
08              ;       3. REASSEMBLE DCUDM AND SYSER.
09              ; THE SIZE OF A DEVICE ERROR MAY BE CHANGED WITHOUT TOO MUCH EFF
10              ; TO DO THIS, DO THE FOLLOWING:
11              ;       1. CHANGE THE DEVICE ERROR Q CONTROL VALUES.
12              ;       2. CHANGE THE DEVICE ERROR TEMPLATE.
13              ;       3. REASSEMBLE DCUDM AND SYSER.
14              ; NEW QUEUES CAN BE ADDED TO THIS AREA.
15              ; PUT AND GET ROUTINES FOR THE NEW QUEUE MUST BE MADE. FOR THE R
16              ; DO THE FOLLOWING:
17              ;       1. CHANGE THE SYSTEM Q CONTROL TEMPLATE.  Q SPACE MAY BE
18              ;               NECESSARY. OTHER Q SPACE MAY NEED TO BE REALLOCAT
19              ;       2. REASSEMBLE DCUDM AND SYSER.
20              ; NOTE: ALL MANIPULATIONS OF EVENTS IN THE Q AREA SHOULD PERFORM
21              ;       SANITY CHECK(SEE EXISTING CHECK) AFTER MAPPING IN THE Q A
22              ;       ALL INITIALIZATION OF THE QUEUE AREA SHOULD BE IN ONE PLA
23              ;       IN SINIT.
24
25              ;SYSTEM Q CONTROL HEADER TEMPLATE(MUST START AT 66000).
26      000000 QQCHD = 0                        ;Q CONTROL HEADER.
27                      ;DEVICE ERROR CONTROL OFFSETS.
28      000001 QQDE0 = 1                        ;EVENT SLOT 0.
29      000002 QQDOE = 2                        ;OVERFLOW SLOT.
30      000003 QQDES = 3                        ;EVENT SIZE.
31      000004 QQDOS = 4                        ;OVERFLOW EVENT SIZE.
32      000005 QQDEM = 5                        ;MAXIMUM SLOT FOR DEVICE ERROR
33      000006 QQDFP = 6                        ;NEXT FREE SLOT.
34      000007 QQDBP = 7                        ;OLDEST LOGGED ERROR STILL IN Q.
35      000010 QQDOF = 10                       ;DEVICE ERROR OVERFLOW INDICATOR.
36
37              ;Q CONTROL VALUES.
38      066000 QCHD = 66000                     ;Q MUST START HERE(SANITY CHECK).
39      066400 QEND = 66400                     ;1 WORD BEYOND THE Q SPACE ALLOCA
40      001577 QMSK =  1577                     ;INTERRUPT MASK FOR Q.
41
42              ;EVENT CODES FOR SYSTEM EVENTS.
43              ;QDEC = ?LDER                   ;DEVICE ERROR EVENT CODE IS ?LDER
44
45              ;DEVICE ERROR Q CONTROL VALUES(QD = QUEUE FOR DEVICE ERRORS).
46      066014 QDE0 = 66014                     ;STARTING ADDRESS OF Q.
47      066374 QDOE = 66374                     ;STARTING ADDRESS OF OVERFLOW.
48      000004 QDES = 4                         ;ENTRY SIZE.
49      000001 QDOS = 1                         ;OVERFLOW EVENT SIZE.
50      000073 QDEM = 73                        ;MAXIMUM SLOT #.
51
52              ;DEVICE ERROR TEMPLATE(QDES = 4).
53      000000 QQDEV = 0                        ;DEVICE CODE.
54      000001 QQUNT = 1                        ;UNIT.
55      000002 QQSTS = 2                        ;ERROR STATUS.
56      000003 QQRTY = 3                        ;RETRY COUNT.
57
58              ;DEVICE OVERFLOW EVENT TEMPLATE(QDOS=1).
59      000000 QQDON = 0                        ;OFFSET HOLDS # OF EVENTS LOST I
60                                              ;THE SYSTEM WAS BOOTED.
```

INTENTIALLY BLANK

```
01                ;
02                ; SYSTEM FATAL ERROR CODES
03                ;
04                ;NOTE THAT THE SYSTEM FATAL ERROR CODES ARE NOW DEFINED IN A
05                ;       SEPERATE PARAMETER FILE -- AOS.PANICS.SR
06                ;
07                ;
08
09
10
11
12                ; SYSTEM PEND KEYS
13
14        000001 .DUSR    SKTRM=  1.        ; WAIT FOR SON'S TERM
15        000002 .DUSR    SKOOL=  2.        ; OVERLAY LOADING
16        000003 .DUSR    SKOOV=  3.        ; OVERLAY CHAIN NEEDS TO GROW
17        000004 .DUSR    SKOOM=  4.        ; AOS TASK NEEDS MEM
18        000005 .DUSR    SKSWP=  5.        ; WAITING FOR SWAP OF PROCESS TO FINISH
19        000006 .DUSR    SKSIO=  6.        ; SHARED READ GROSS WAIT KEY
20        000007 .DUSR    SKBUF=  7.        ; BASE LEVEL NEEDS SYSTEM BUFFER
21        000010 .DUSR    SKNQP=  8.        ; WAITING FOR I/O NQ BLOCK
22        000011 .DUSR    SKA01=  9.        ; PERFORMANCE EVALUATION
23        000012 .DUSR    SKA02=  10.       ; PERFORMANCE EVALUATION
24        000040 .DUSR    SKSYN=  40        ;RESERVE BLOCK OF 32. KEYS FOR
25                                          ;SYNC ...LINE SUPPORT
```

```
!0009  PARS
01                ;
02                ; IPC SYSTEM DATA BASES
03
04                ; OUTSTANDING RECEIVE ENTRY (REV 3 FORMAT)

        000000 .DUSR    OLNK=   0         ; LINK WORD (MUST BE OFFSET ZERO)
        000001 .DUSR    OOPH=   OLNK+1    ; ORIGIN PORT NO. HIGH (HID/PID)
08      000002 .DUSR    OOPL=   OOPH+1    ; ORIGIN PORT NO. LOW (16 BIT PORT #)
09      000003 .DUSR    ODPN=   OOPL+1    ; DESTINATION PORT NUMBER
10      000004 .DUSR    OLTH=   ODPN+1    ; BUFFER LENGTH (IN WORDS)
11      000005 .DUSR    OADDR=  OLTH+1    ; POINTER TO USER BUFFER
12      000006 .DUSR    OBUFH=  OADDR+1   ; POINTER TO USER BUFFER HEADER
13                                        ;   100 = 1 => GHOST MADE ?IREC CALL
14      000007 .DUSR    OTCB=   OBUFH+1   ; USER TCB ADDRESS
15      000010 .DUSR    OSFL=   OTCB+1    ; SYSTEM FLAG WORD
16
17      000011 .DUSR    ORLTH=  OSFL-OLNK+1     ;LENGTH OF ENTRY
18
19                ; SPOOL FILE DIRECTORY BLOCK HEADER
20
21      000000 .DUSR    ISNE=   0         ;NUMBER OF ENTRIES IN DIRECTORY
22
23
24                ; SPOOL FILE DIRECTORY ENTRY
25
26      000000 .DUSR    MESFL=  ?ISFL     ;SYSTEM FLAGS
27      000001 .DUSR    MEUFL=  ?IUFL     ;USER FLAGS
28      000002 .DUSR    MEOPH=  ?IOPH     ;ORIGIN PORT NUMBER (HI)
29      000003 .DUSR    MEOPL=  ?IOPL     ;ORIGIN PORT NUMBER (LO)
        000004 .DUSR    MEDPN=  ?IDPN     ;DESTINATION PORT NUMBER
        000005 .DUSR    MELTH=  ?ILTH     ;MESSAGE LENGTH (IN WORDS)
        000006 .DUSR    MEPTR=  MELTH+1   ;WORD POINTER TO MESSAGE TEXT IN FILE
33
34      000007 .DUSR    MDLTH=  MEPTR-MESFL+1     ;LENGTH OF ENTRY
35
36
37                ; SPOOL FILE DIRECTORY ENTRY CONSTANTS
38
39      000043 .DUSR    ISMNE=  (256./MDLTH)-1    ;MAX NUMBER OF ENTRIES ALLOWED
40
```

```
01
02              ; PERIPHERAL MANAGER I/O CONTROL BITS
03
04              ; "IN:" MEANS THE BIT IS INPUT FROM A USER
05              ; "OUT:" MEANS THE BIT IS OUTPUT FROM THE PMGR
06              ; "XXX/YYY" MEANS IF BIT=0 FUNCTION=XXX, IF BIT=1 FUNCTION=YYY
07
08      000000 .DUSR    RWIOG=  0.      ;IN: PRIMARY/GHOST BUFFER CONTEXT
09                                      ;OUT: (RESERVED FOR FUTURE USE)
10      000001 .DUSR    RWERB=  1.      ;IN: READ DONT ECHO SINGLE CHAR DELIM
11                                      ;OUT: ERROR BIT
12      000002 .DUSR    RWDRP=  2.      ;IN: READ DROP TYPED-AHEAD BYTES
13                                      ;OUT: READ ENDED WITH A 2 CHAR DELIM
14      000003 .DUSR    RWRNE=  3.      ;IN: ECHO/NO ECHO ON THIS READ OR
15                                      ;IN: NORMAL/FORCED WRITE (CLEAR ^O)
16                                      ;OUT: (RESERVED FOR FUTURE USE)
17      000004 .DUSR    RWBTF=  4.      ;IN: TEXT/BINARY I/O FLAG
18                                      ;OUT: (RESERVED FOR FUTURE USE)
19      000005 .DUSR    RWTTQ=  5.      ;IN: TO DELIMITER/# BYTES SPECIFIED
20                                      ;OUT: (RESERVED FOR FUTURE USE)
21      000006 .DUSR    RWPRF=  6.      ;IN: NORMAL/PRIORITY READ REQUEST
22                                      ;OUT: (RESERVED FOR FUTURE USE)
23              ;               7.-15.  ;IN: READ & WRITE (MAX) # BYTES
24              ;                       ;OUT: # BYTES READ OR WRITTEN, INCL DELIM
25
26              ;       BIT POINTERS
27      000020 .DUSR    BWIOG=  ?IUFL*16.+RWIOG
28      000021 .DUSR    BWERR=  ?IUFL*16.+RWERB
29      000022 .DUSR    BWDRP=  ?IUFL*16.+RWDRP
30      000023 .DUSR    BWRNE=  ?IUFL*16.+RWRNE
31      000024 .DUSR    BWBTF=  ?IUFL*16.+RWBTF
32      000025 .DUSR    BWTTQ=  ?IUFL*16.+RWTTQ
33      000026 .DUSR    BWPRF=  ?IUFL*16.+RWPRF
```

```
!0011  PARS
01                ; PERIPHERAL MANAGER EXTENDED I/O CONTROL PACKET
02
03                ;FORMAT OF EXTENDED PACKET (3 WORD IPC MESSAGE)

                  ;0 - <COL><ROW>
                  ;1 - USER OPTIONS  AND <EDIT COL>        SEE BELOW
                  ;2 - BUFFER BYTE ADDRESS
08
09                ; THE FOLLOWING ARE USER OPTIONS:
10                     BITS
11          000000 SENR=0   ; DISABLE REDISPLAY ON THIS SCREEN-EDIT READ
12          000001 SEIO=1   ; THIS REQ IS FOR SCREEN-EDIT READ
13          000002 SEPC=2   ; POSITION CURSOR BEFORE I/O
14          000003 SERC=3   ; RETURN CURSOR POSITION
15                ;
16                ;
17                ;
18                ;
19                ;
20                ; BITS 9-15 SPECIFY EDIT CURSOR POSITION [IF 0B(SENR)]
21
22
23
24                ; MASKS FOR EXTRACTING PROPER INFO FROM WORD 1 OF PACKET
25
26          170000 SCOPT    =1B(SENR)+1B(SEIO)+1B(SEPC)+1B(SERC) ; VALID OPTIONS
27          000177 SCPM     =177     ; EDIT POSITION MASK
28
```

```
!0012  PARS
01
02
03
04              ; PERIPHERAL MANAGER COMMAND CODES (BITS# 4,5,6,7)
05
06      000400 .DUSR    CMOFF=  400     ;OFFSET FOR COMMANDS
07
08      000000 .DUSR    PMCOP=  0*CMOFF ;OPEN
09      000400 .DUSR    PMCCL=  1*CMOFF ;CLOSE
10      001000 .DUSR    PMCAS=  2*CMOFF ;ASSIGN
11      001400 .DUSR    PMCRL=  3*CMOFF ;DEASSIGN
12      002000 .DUSR    PMCAC=  4*CMOFF ;ASSIGN A CONSOLE (PROCESS CREATE)
13      002400 .DUSR    PMCPT=  5*CMOFF ;PROCESS TERMINATE
14      003000 .DUSR    PMCPC=  6*CMOFF ;PROCESS CHAIN
15      003400 .DUSR    PMCGT=  7*CMOFF ;GET DELIMITER TABLE (ONE DTBL)
16      004000 .DUSR    PMCST=  10*CMOFF ;SET DELIM TABLE (ANY OR ALL DTBL'S)
17      004400 .DUSR    PMCGC=  11*CMOFF ;GET DEVICE CHARACTERISTICS
18      005000 .DUSR    PMCSC=  12*CMOFF ;SET DEVICE CHARACTERISTICS
19      005400 .DUSR    PMCSM=  13*CMOFF ;SEND MESSAGE TO DEVICE (NOT RETURNED)
20      006000 .DUSR    PMCRT=  14*CMOFF ;TERMINATE REQUEST(S) OF A TASK
21      006400 .DUSR    PMCTO=  15*CMOFF ;SET TIME-OUT CONSTANTS
22              ;               16      ;(RESERVED FOR FUTURE USE)
23              ;               17      ;(RESERVED FOR FUTURE USE)
24
25              ; PERIPHERAL MANAGER COMMAND CONTROL BITS
26
27      000010 .DUSR    DTBLO=  8.      ;GET/SET OUTPUT DELIMITER TABLE
28      000011 .DUSR    DTBLI=  9.      ;GET/SET INPUT DELIMITER TABLE
29      000012 .DUSR    DTBLP=  10.     ;GET/SET PRIORITY READ DELIM TABLE
30      000013 .DUSR    CHBDF=  11.     ;GET/SET DEFAULT CHARACTERISTICS
31
32              ;       BIT POINTERS
33      000030 .DUSR    BWBLO=  ?IUFL*16.+DTBLO
34      000031 .DUSR    BWBLI=  ?IUFL*16.+DTBLI
35      000032 .DUSR    BWBLP=  ?IUFL*16.+DTBLP
36      000033 .DUSR    BWBDF=  ?IUFL*16.+CHBDF
```

```
!0013  PARS
01
02              ; PERIPHERAL INFORMATION BLOCK (PIB)
03
        000000 .DUSR    PIBOD=  0         ;OUTPUT DEVICE CODE (RH) - LINE # (LH)
        000001 .DUSR    PIBID=  PIBOD+1   ;INPUT DEVICE CODE (RH) - LINE # (LH)
        000002 .DUSR    PIBS1=  PIBID+1   ;CURRENT STATUS WORD #1
07      000003 .DUSR    PIBS2=  PIBS1+1   ;CURRENT STATUS WORD #2
08      000004 .DUSR    PIBLK=  PIBS2+1   ;LINKS PIBS FOR THIS OWNER
09      000005 .DUSR    PIBIS=  PIBLK+1   ;INPUT START ROUTINE
10      000006 .DUSR    PIBOS=  PIBIS+1   ;OUTPUT START ROUTINE
11      000007 .DUSR    PIBQ=   PIBOS+1   ;BLOCKED PIB QUEUE LINK
12      000010 .DUSR    PIBRC=  PIBQ+1    ;RESTART CODES (LH=INPUT, RH=OUTPUT)
13      000011 .DUSR    PIBCQ=  PIBRC+1   ;CONTROL COMMAND REQUEST QUEUE
14      000012 .DUSR    PIBRQ=  PIBCQ+1   ;READ REQUEST QUEUE
15      000013 .DUSR    PIBWQ=  PIBRQ+1   ;WRITE REQUEST QUEUE
16              ; THE FOLLOWING 7 WORDS MUST STAY IN THIS ORDER :
17      000014 .DUSR    PIBSS=  PIBWQ+1   ;STATE SAVE AREA (SSA) ADR OR 0
18      000015 .DUSR    PIBOW=  PIBSS+1   ;OWNER'S PID (180=1 =IMPLICIT ASSIGN)
19      000016 .DUSR    PIBDT=  PIBOW+1   ;WORD POINTER TO INPUT DELIM TABLE
20      000017 .DUSR    PIBPT=  PIBDT+1   ;WORD POINTER TO PRI READ DELIM TABLE
21      000020 .DUSR    POBDT=  PIBPT+1   ;WORD POINTER TO OUTPUT DELIM TABLE
22      000021 .DUSR    PIBTC=  POBDT+1   ;TIME-OUT CONSTANT
23      000022 .DUSR    PIBOC=  PIBTC+1   ;DEVICE OPEN COUNTER
24              ; THE FOLLOWING THREE WORDS MUST STAY IN THIS ORDER :
25      000023 .DUSR    PIBC1=  PIBOC+1   ;DEVICE CHARACTERISTICS - WORD ONE
26      000024 .DUSR    PIBC2=  PIBC1+1   ;DEVICE CHARACTERISTICS - WORD TWO
27      000025 .DUSR    PIBC3=  PIBC2+1   ;LINES/PAGE (LH), BYTES/LINE (RH)
28              ; THE FOLLOWING FOUR WORDS MUST STAY IN THIS ORDER :
29      000026 .DUSR    PIBRH=  PIBC3+1   ;READ PORT NUMBER (HI)
        000027 .DUSR    PIBRL=  PIBRH+1   ;READ PORT NUMBER (LO)
        000030 .DUSR    PIBWH=  PIBRL+1   ;WRITE PORT NUMBER (HI)
32      000031 .DUSR    PIBWL=  PIBWH+1   ;WRITE PORT NUMBER (LO)
33      000032 .DUSR    PIBD1=  PIBWL+1   ;DEFAULT DEVICE CHAR'S - WORD ONE
34      000033 .DUSR    PIBD2=  PIBD1+1   ;DEFAULT DEVICE CHAR'S - WORD TWO
35      000034 .DUSR    PIBD3=  PIBD2+1   ;LINES/PAGE (LH), BYTES/LINE (RH)
36              ; INPUT BUFFER INFO (BYTE VALUES UNLESS SPECIFIED)
37      000035 .DUSR    IBBEG=  PIBD3+1   ;START OF BUFFER
38      000036 .DUSR    IBEND=  IBBEG+1   ;END OF BUFFER
39      000037 .DUSR    IBCIP=  IBEND+1   ;CURRENT INSERT POINTER
40      000040 .DUSR    IBCRP=  IBCIP+1   ;CURRENT REMOVE POINTER
41      000041 .DUSR    IBMAP=  IBCRP+1   ;PAGE 66 MAP WORD FOR INPUT BUFFER
42      000042 .DUSR    PIBSZ=  IBMAP+1   ;SIZE FOR INPUT BUFFER (=0 => NO INPUT)
43              ; OUTPUT BUFFER INFO (BYTE VALUES UNLESS SPECIFIED)
44      000043 .DUSR    OBBEG=  PIBSZ+1   ;START OF BUFFER
45      000044 .DUSR    OBEND=  OBBEG+1   ;END OF BUFFER
46      000045 .DUSR    OBCIP=  OBEND+1   ;CURRENT INSERT POINTER
47      000046 .DUSR    OBCRP=  OBCIP+1   ;CURRENT REMOVE POINTER
48      000047 .DUSR    OBMAP=  OBCRP+1   ;PAGE 66 MAP WORD FOR OUTPUT BUFFER
49      000050 .DUSR    POBSZ=  OBMAP+1   ;SIZE FOR OUTPUT BUF (=0 => NO OUTPUT)
50              ; COUNTERS & CONSTANTS
51      000051 .DUSR    PIBCC=  POBSZ+1   ;COLUMN COUNTER
52      000052 .DUSR    PIBLC=  PIBCC+1   ;LINE COUNTER
53      000053 .DUSR    PIBSC=  PIBLC+1   ;READ STARTING COLUMN COUNTER
54      000054 .DUSR    PIBTO=  PIBSC+1   ;OUTPUT TIME-OUT TIMER
        000055 .DUSR    PIBTI=  PIBTO+1   ;INPUT TIME-OUT TIMER
        000056 .DUSR    PIBTD=  PIBTI+1   ;DEFAULT TIME-OUT CONSTANT
        000057 .DUSR    PIBCL=  PIBTD+1   ;DEVICE CLEAR ROUTINE
58
59      000060 .DUSR    PIBLN=  PIBCL+1   ;LENGTH OF PIB
```

```
0014  PARS
01
02              ; PERIPHERAL STATUS BITS (IN PIBS1)
03
04      000000 .DUSR    TSCIP=  0.      ;DEVICE OUTPUT ACTIVE (PHYSICAL I/O)
05      000001 .DUSR    TSOS=   1.      ;OUTPUT SUSPENDED (BY ^O)
06      000002 .DUSR    TSOH=   2.      ;OUTPUT HELD (BY ^S)
07      000003 .DUSR    TSOD=   3.      ;OUTPUT DONE (UNBLOCKED)
08      000004 .DUSR    TSBOR=  4.      ;BLOCKED ON OUTPUT ROOM
09      000005 .DUSR    TSBCW=  5.      ;BLOCKED ON CONNECT(WRITE)
10      000006 .DUSR    TSID=   6.      ;INPUT DONE (UNBLOCKED)
11      000007 .DUSR    TSBIT=  7.      ;BLOCK ON INPUT OF TERMINATOR
12      000010 .DUSR    TSBIB=  8.      ;BLOCKED ON INPUT OF # OF BYTES
13      000011 .DUSR    TSIBM=  9.      ;INPUT IN BINARY MODE
14      000012 .DUSR    TSPCP=  10.     ;PROCESS CONSOLE DEVICE
15      000013 .DUSR    TSREO=  11.     ;READ ECHO BLOCKED ON OUTPUT ROOM
16      000014 .DUSR    TSLCS=  12.     ;LITERAL CHARACTER SEEN
17      000015 .DUSR    TSICS=  13.     ;INTERUPT CHARACTER SEEN
18      000016 .DUSR    TSRM=   14.     ;BLOCKED ON READ MEMORY AVAILABILITY
19      000017 .DUSR    TSWM=   15.     ;BLOCKED ON WRITE MEMORY AVAILABILITY
20
21              ; PIBS1 BIT POINTERS
22
23      000040 .DUSR    BTSOP=  PIBS1*16.+TSCIP
24      000041 .DUSR    BTSOS=  PIBS1*16.+TSOS
25      000042 .DUSR    BTSOH=  PIBS1*16.+TSOH
26      000043 .DUSR    BTSOD=  PIBS1*16.+TSOD
27      000044 .DUSR    BTSBO=  PIBS1*16.+TSBOR
28      000045 .DUSR    BTSCW=  PIBS1*16.+TSBCW  BSCW
29      000046 .DUSR    BTSID=  PIBS1*16.+TSID   BTSI
30      000047 .DUSR    BTSBT=  PIBS1*16.+TSBIT
31      000050 .DUSR    BTSBB=  PIBS1*16.+TSBIB
32      000051 .DUSR    BTSIB=  PIBS1*16.+TSIBM
33      000052 .DUSR    BTSPC=  PIBS1*16.+TSPCP
34      000053 .DUSR    BTSRE=  PIBS1*16.+TSREO
35      000054 .DUSR    BTSLS=  PIBS1*16.+TSLCS
36      000055 .DUSR    BTSIC=  PIBS1*16.+TSICS
37      000056 .DUSR    BTSRM=  PIBS1*16.+TSRM
38      000057 .DUSR    BTSWM=  PIBS1*16.+TSWM
39
40              ; PIBS1 MASKS
41
42      000040 .DUSR    TSCON=  1B(TSPCP)                               ;CONSOLE DEVICE M
43      000620 .DUSR    IBMSK=  1B(TSBIT)+1B(TSBIB)+1B(TSREO)           ;INPUT BLOCKED MA
44      006000 .DUSR    OBMSK=  1B(TSBOR)+1B(TSBCW)                     ;OUTPUT BLOCKED M
45      011000 .DUSR    DNMSK=  1B(TSID)+1B(TSOD)                       ;I/O DONE MASK
```

```
!0015  PARS
01
02              ; PERIPHERAL STATUS BITS (IN PIBS2)
03
        000000 .DUSR    TSPRE=  0.      ;PRIORITY READ ENQUEUED
        000001 .DUSR    TSEBA=  1.      ;RUBOUT ECHO BACKARROW ONLY
        000002 .DUSR    TSNEL=  2.      ;ECHO ^X AS ^X, NL ONLY
07      000003 .DUSR    TSCDR=  3.      ;DEVICE IS A CARD READER
08      000004 .DUSR    TSWCN=  4.      ;FIRST WRITE-WAIT FOR CONNECT
09      000005 .DUSR    TSLDR=  5.      ;SKIP LEADER MODE
10      000006 .DUSR    TXMT=   6.      ;XMIT OFF SYNC STATUS FOR ALM
11      000007 .DUSR    TIFK=   7.      ;INPUT FUNCTION KEY SEEN
12      000010 .DUSR    TSPFL=  8.      ;POWERFAIL OCCURRED ON MODEM LINE
13      000011 .DUSR    TSDCS=  9.      ; DELAYED ^S (FOR ?SEND PROCESSING)
14      000012 .DUSR    TSDPM=  10.     ; DELAYED PAGE-MODE (DITTO)
15      000013 .DUSR    TSPAR=  11.     ; PARITY ERROR OCCURRED (ALM LINE)
16              ;               12.
17              ;               13.
18      000016 .DUSR    TSDSR=  14.     ;CONNECT FOR MODEM CONTROLLED LINE (DSR)
19              ;               15.
20
21              ; PIBS2 BIT POINTERS
22
23      000060 .DUSR    BTSPR=  PIBS2*16.+TSPRE
24      000061 .DUSR    BTSBA=  PIBS2*16.+TSEBA
25      000062 .DUSR    BTSEL=  PIBS2*16.+TSNEL
26      000063 .DUSR    BTCDR=  PIBS2*16.+TSCDR
27      000064 .DUSR    BTWCN=  PIBS2*16.+TSWCN
28      000065 .DUSR    BTLDR=  PIBS2*16.+TSLDR
29      000066 .DUSR    BTXMT=  PIBS2*16.+TXMT
        000067 .DUSR    BTIFK=  PIBS2*16.+TIFK
        000070 .DUSR    BTPFL=  PIBS2*16.+TSPFL
        000071 .DUSR    BTDCS=  PIBS2*16.+TSDCS
33      000072 .DUSR    BTDPM=  PIBS2*16.+TSDPM
34      000073 .DUSR    BTPAR=  PIBS2*16.+TSPAR
35              ;
36              ;
37      000076 .DUSR    BTCON=  PIBS2*16.+TSDSR
38              ;
```

```
0016  PARS
 1
 2                ; DEVICE CHARACTERISTICS BIT POINTERS
 3                ;       THE BIT OFFSETS (?CXXX) ARE DEFINED IN PARU
 4                ;       SO THAT THE USERS CAN PARAMETERIZE THEM
 5
 6       000460 .DUSR    BCST=   PIBC1*16.+?CST
 7       000461 .DUSR    BCSFF=  PIBC1*16.+?CSFF
 8       000462 .DUSR    BCEPI=  PIBC1*16.+?CEPI
 9       000463 .DUSR    BC8BT=  PIBC1*16.+?C8BT
10       000464 .DUSR    BCSPO=  PIBC1*16.+?CSPO
11       000465 .DUSR    BCRAF=  PIBC1*16.+?CRAF
12       000466 .DUSR    BCRAT=  PIBC1*16.+?CRAT
13       000467 .DUSR    BCRAC=  PIBC1*16.+?CRAC
14       000470 .DUSR    BCNAS=  PIBC1*16.+?CNAS
15       000471 .DUSR    BCOTT=  PIBC1*16.+?COTT
16       000472 .DUSR    BCEOL=  PIBC1*16.+?CEOL
17       000473 .DUSR    BCUCO=  PIBC1*16.+?CUCO
18       000475 .DUSR    BCFF=   PIBC1*16.+?CFF
19       000474 .DUSR    BCLT=   PIBC1*16.+?CLT
20
21       000500 .DUSR    BCULC=  PIBC2*16.+?CULC
22       000501 .DUSR    BCPM=   PIBC2*16.+?CPM
23       000502 .DUSR    BCNRM=  PIBC2*16.+?CNRM
24       000503 .DUSR    BCMOD=  PIBC2*16.+?CMOD
25       000510 .DUSR    BCETO=  PIBC2*16.+?CTO
26       000511 .DUSR    BCSPP=  PIBC2*16.+?CTSP
27       000513 .DUSR    BCESC=  PIBC2*16.+?CESC
28       000512 .DUSR    BCPBN=  PIBC2*16.+?CPBN
29       000514 .DUSR    BCWRP=  PIBC2*16.+?CWRP
30       000515 .DUSR    BCFKT=  PIBC2*16.+?CFKT
31       000516 .DUSR    BCNNL=  PIBC2*16.+?CNNL
32                ;       (RESERVED FOR FUTURE USE)
```

```
!0017. PARS
01
02              ; DEVICE INTERRUPT MASKS
03
0         000003 .DUSR    MKTTO=  1B15+1B14       ; TTO
          000003 .DUSR    MKTTI=  MKTTO           ; TTI
          000003 .DUSR    MKQTY=  MKTTI           ; QTY
07        000007 .DUSR    MKPTP=  1B13+MKTTI      ; PTP
08        000017 .DUSR    MKMCA=  1B12+MKPTP      ; MCA
09        000017 .DUSR    MKPLT=  MKMCA           ; PLT
10        000010 .DUSR    MKLPT=  MKPLT-MKPTP     ; LPT
11        000217 .DUSR    MKDPO=  1B08+MKMCA      ; DPO
12        000217 .DUSR    MKDPI=  MKDPO           ; DPI
13        000517 .DUSR    MKDKP=  157+1B9+MKMCA   ; MHD
14        000377 .DUSR    MKMUX=  377             ; MUX(SLM AND ALM)
15        000517 .DUSR    MKDSK=  MKDKP           ; FHD
16        001517 .DUSR    MKIPB=  1B06+MKDSK      ; IPB
17        001537 .DUSR    MKPTR=  1B11+MKIPB      ; PTR
18        001577 .DUSR    MKCDR=  1B10+MKPTR      ; CDR
19        001577 .DUSR    MKMTA=  MKCDR           ; MTA
20        001577 .DUSR    MKCAS=  MKMTA           ; CAS
21        007777 .DUSR    MKDCU=  7777            ; DCU
22        003777 .DUSR    MKIOP=  3777            ; IOP
23        000004 .DUSR    MKRTC=  1B13            ; RTC
24        001000 .DUSR    MKPIT=  1B6             ; INTERVAL TIMER
25
26
27              ; USER DEVICE TABLE OFFSETS
28
29              ; DEFINITION OF OFFSETS
          000000 .DUSR    UIMPS=  0               ;ADDRESS OF 'MAPST'
          000001 .DUSR    UIMSK=  1               ;DEVICE INTERRUPT MASK
          000002 .DUSR    UIPTB=  2               ;PROCESS TABLE ADDRESS
33        000003 .DUSR    UIDEX=  3               ;ADDRESS OF 'UDEX' (ENTRY POINT
34                                                ;FOR ALL USER INTERRUPTS)
35        000004 .DUSR    UIDCH=  4               ;LH = # OF DCH SLOTS
36                                                ;RH = 1ST DCH SLOT #
37        000005 .DUSR    UIDCD=  5               ;DEVICE CODE
38        000006 .DUSR    UILKF=  6               ;FORWARD LINK (OFF PICHN)
39        000007 .DUSR    UILKB=  7               ;BACKWARD LINK (OFF PICHN)
40        000010 .DUSR    UIDIS=  10              ;USER ADDRESS OF HIS INTERRUPT
41                                                ;SERVICE ROUTINE
42        000011 .DUSR    UIDCT=  11              ;USER DCT ADDRESS
43        000012 .DUSR    UIPRS=  12              ;POWERFAIL RESTART ROUTINE ADDRES
44
45        000013 .DUSR    UILTH=  UIPRS+1
46
47
```

```
01                      ;DEFINE HARDWARE PAGE FAULT OFFSETS
02
03
04                      ; DEFINE THE FAULT BLOCK OFFSETS
05              000000 .DUSR    FBVLH=  0       ; VALIDITY BITS FOR PAGES 0-15
06              000001 .DUSR    FBVLL=  1       ; VALIDITY BITS FOR PAGES 16-32
07              000000 .DUSR    FBCTX=  0       ; CONTEXT BLOCK START
08              000022 .DUSR    FBFLT=  22      ; FLOATING POINT STATE
09
10              000044 .DUSR    FBLGN=  44      ; SIZE OF THE FAULT BLOCK
11
12
13                      ; DEFINE THE RELAVENT OFFSETS IN THE CONTEXT BLOCK
14              000007 .DUSR    FLSTS=  7       ; STATUS WORD
15              000013 .DUSR    FLPAGE= 13      ; PAGE THAT FAULTED
16              000015 .DUSR    FLAC0=  15      ; AC0
17              000016 .DUSR    FLAC1=  16      ; AC1
18              000017 .DUSR    FLAC2=  17      ; AC2
19              000020 .DUSR    FLAC3=  20      ; AC3
20              000021 .DUSR    FLGR3=  21      ; GR3 (MICROCODE PC)
21
22
23                      ; STATUS WORD BIT
24              010000 .DUSR    FLFSM=  1B3     ; FAULT IS IN SYSTEM MAP (MAP 6)
25
26                      ; STATUS WORD BIT POINTER
27              000163 .DUSR    BFFSM=  FLSTS*16.+3
```

```
!0019  PARS
01                ;
02                ;                    AOS PROCESS TABLE
03                ;

                  ; PROCESS TABLE CONTAINS TWO AREAS- ONE IS PERMANENT AND ONE
                  ; SWAPS WITH THE CORE IMAGE
07                ; PEXTN IN THE PERMANENT AREA POINTS TO THE EXTENSION
08
09                ; ALL PARAMETERS UNIQUE TO THE E-200 OR E-500 EXIST AT
10                ; THE END OF THE PARAMETER GROUP IN ORDER TO REDUCE THE
11                ; NUMBER OF CONDITIONAL ASSEMBLIES NEEDED.
12
13
14                ; PROCESS TABLE OFFSETS
15                ; OFFSETS 4-10 MUST BE COMMON BETWEEN ALL ENTRIES
16
17
18          000000 .DUSR   PSELF=  0        ;ADDR OF PTBL (FOR @BIT INST...)
19
20                ; DEFINE THE PARAMETERS USED FOR THE PROCESS TREE
21          000001 .DUSR   PDAD=   1        ;PTBL ADDR OF FATHER
22          000002 .DUSR   PSONP=  2        ;SON POINTER TO SON LIST
23          000003 .DUSR   PSONL=  3        ;SON LINK WD FOR FATHER'S SON LIST
24
25                ; COMMON ENTRIES USED BY THE SCHEDULER
26
27          000004 .DUSR   PSTAT=  4        ;STATUS OF THIS DATABASE
28          000005 .DUSR   PLNK=   5        ;ACTIVE CHAINFORWARD LINK
29          000006 .DUSR   PBLNK=  6        ;ACTIVE CHAINBACK LINK
            000007 .DUSR   PNQF=   7        ;PRI ENQUE FACTOR
            000010 .DUSR   PPC=    10       ;CONTROL ADDR WHEN SCHEDULED
32          000011 .DUSR   PKEY=   11       ;UNPEND KEY
33
34                ; END OF COMMON AREA BETWEEN CB AND PTBL
35
36                ; DEFINE THE PROCESS FLAG WORDS
37                ; THESE WORDS CONTAIN FLAGS THAT DESCRIBE BOTH STATIC
38                ; AND DYNAMIC PROCESS CHARACTERISTICS
39
40          000012 .DUSR   PFLAG=  PKEY+1   ;FLAGS
41          000013 .DUSR   PFLG2=  PFLAG+1  ;FLAG WORD 2
42          000014 .DUSR   PFLG3=  PFLG2+1  ;FLAG WORD 3
43          000015 .DUSR   PFLG4=  PFLG3+1  ;FLAG WORD 4
44
45                ; DEFINE A POINTER TO THE SWAPPABLE PORTION OF THE PROCESS TABLE
46          000016 .DUSR   PEXTN=  PFLG4+1  ;ADDR OF EXTENSION
47
48          000017 .DUSR   PPRI=   PEXTN+1  ;FATHER ASSIGNED PRI FACTOR
49
50                ; DEFINE THE IPC CONTROL PARAMETERS
51          000020 .DUSR   PIORR=  PPRI+1   ;BLOCKED RECEIVE REQUEST CHAIN
52          000021 .DUSR   PIPCS=  PIORR+1  ;SPOOL FILE
53
54
                  ; NOTE-  GROUPS DEFINING PRIMARY AND GHOST AREAS MUST FOLLOW
            000022 .DUSR   PBLKS=  PIPCS+1  ;# PRIMARY BLKS
            000023 .DUSR   PSHSZ=  PBLKS+1  ;SHARED AREA SIZE
58          000024 .DUSR   PSHIT=  PSHSZ+1  ;SHARED AREA START
59          000025 .DUSR   PAMAP=  PSHIT+1  ;ADDR OF THE PRIMARY MAP
60          000026 .DUSR   PREF1=  PAMAP+1  ;REFERENCED BITS FOR PAGES 0-15
```

```
000027 .DUSR   PR[illegible]   PREF1+1 ;REFERENCED BITS FOR PAGES 16-31
000030 .DUSR   PMOD1=  PREF2+1 ;MODIFIED BITS FOR PAGES 0-15
000031 .DUSR   PMOD2=  PMOD1+1 ;MODIFIED BITS FOR PAGES 16-31
       ; END OF PRIMARY AREA
000032 .DUSR   PGBLS=  PMOD2+1 ;# GHOST IMPURE BLKS
000033 .DUSR   PGSHZ=  PGBLS+1 ;GHOST SHARED SIZE
000034 .DUSR   PGSHS=  PGSHZ+1 ;GHOST SHARED START
000035 .DUSR   PGMAP=  PGSHS+1 ;GHOST MAP ADDR
000036 .DUSR   PGRF1=  PGMAP+1 ;REFERENCED BITS FOR PAGES 0-15
000037 .DUSR   PGRF2=  PGRF1+1 ;REFERENCED BITS FOR PAGES 16-31
000040 .DUSR   PGMD1=  PGRF2+1 ;MODIFIED BITS FOR PAGES 0-15
000041 .DUSR   PGMD2=  PGMD1+1 ;MODIFIED BITS FOR PAGES 16-31
       ; END OF GHOST MEMORY DESCRIPTION

000010 .DUSR   GMDSZ=  PGMAP-PAMAP     ;MAP DISPLACEMENT

000042 .DUSR   PID=    PGMD2+1 ;PROCESS ID ASSIGNED AT CREATE TIME
000043 .DUSR   PRPRV=  PID+1   ;PROC PRIV BITS ASSIGNED BY CREATOR
000044 .DUSR   PRCCB=  PRPRV+1 ;BREAK FILE CCB
000045 .DUSR   PBIOS=  PRCCB+1 ;RESERVED
000046 .DUSR   PCMLK=  PBIOS+1 ;CORE MANAGER ENQUE LINK
000047 .DUSR   PSLEX=  PCMLK+1 ;TIME SLICE EXPONENT
000050 .DUSR   PCRMX=  PSLEX+1 ;MAX # SUN PROCS
000051 .DUSR   PMKEY=  PCRMX+1         ;MEM WAIT FLAG KEY

       ; DEFINE THE PARAMETERS IN THE RESIDENT PTBL NEEDED FOR
       ; THE MANAGEMENT OF ?DELAY CALLS
000052 .DUSR   PDINH=  PMKEY+1 ;DELAY CURRENT INC HIGH WD
000053 .DUSR   PDINL=  PDINH+1 ;DELAY INC LOW WD
000054 .DUSR   PDLNK=  PDINL+1 ;DELAY CHAIN LINK

000055 .DUSR   PKCHR=  PDLNK+1 ;?KWAIT CHAR
000056 .DUSR   PTRGC=  PKCHR+1 ;TARGET SYSTEM CALL COUNTER

000057 .DUSR   PCONH=  PTRGC+1 ;CONSOLE PORT NUMBER (HI)
000060 .DUSR   PCONL=  PCONH+1 ;CONSOLE PORT NUMBER (LO)
000061 .DUSR   PHASH=  PCONL+1 ;PROCESS NAME HASH VALUE
000062 .DUSR   PTUP1=  PHASH+1 ;FIRST TCB UNPEND REQ WD
000063 .DUSR   PTUP2=  PTUP1+1 ;SECOND TCB UNPEND REQ WD
000064 .DUSR   PICHN=  PTUP2+1 ;PTR TO USER DCT LINKED LIST
000065 .DUSR   PINSU=  PICHN+1 ;FLAG FOR EITHER GHOST OR PRIM IN SCHED M
000066 .DUSR   PSSEL=  PINSU+1 ;# OF SUB-SLICES USED SINCE PUT ON ELQUE
000067 .DUSR   PHLNK=  PSSEL+1 ;HISTOGRAM LINKAGE WORD
000070 .DUSR   PSWBN=  PHLNK+1 ;SWAP FILE BLOCK #
000071 .DUSR   PSHUC=  PSWBN+1 ;TOTAL SHARED SLOTS IN USE
000072 .DUSR   PSHFC=  PSHUC+1 ;# SHARED SLOTS THAT HAD TO BE LOADED
                               ;DURING THE LAST SWAP-IN

       ; THIS IS THE END OF THE COMMON RESIDENT PTBL AREA
000072 .DUSR   PRCEN=  PSHFC   ;DEFINES LAST COMMON LOCATION
```

```
!0021  PARS
01
02
03                ; DEMAND PAGED SYSTEM -- UNIQUE AREA
04
05        000000          .IFN    NEWDP
06                .DUSR   PMASS=  PRCEN+1 ;SCHED SIZE
07                .DUSP   PCMAP=  PMASS+1 ;@CMAP AT TIME OF FAULT
08                .DUSR   PVEXT=  PCMAP+1 ;VIRTUAL EXTENDER ADDRESS
09                .DUSR   PREXT=  PVEXT+1 ;RESIDENT EXTENDER ADDRESS
10
11                .DUSR   PLN=    PREXT+1
12                        .ENDC
13
14        000001          .IFE    NEWDP
15
16                ; E-200 UNIQUE AREA
17
18        000073 .DUSR    PMASS=  PRCEN+1 ;SCHED SIZE
19        000074 .DUSR    PNSWP=  PMASS+1 ;"DO NOT SWAP" COUNTER
20                ;
21                ; IN ORDER TO ALLOW UTILITIES LIKE ADA, SYSDMP, ETC TO WORK ACRO
22                ; REV'S PLEASE LEAVE PVEXT AT OFFSET 75.....
23                ;
24        000075 .DUSR    PVEXT=  PNSWP+1 ;VIRTUAL EXTENDER ADDR
25        000076 .DUSR    PREXT=  PVEXT+1 ;RESIDENT EXTENDER ADDRESS
26
27        000077 .DUSR    PLN=    PREXT+1 ;SIZE OF THE RESIDENT PORTION
28
29                        .ENDC
```

```
01                      ;
02                      ;RESIDENT PROCESS TABLE EXTENSION OFFSETS
03                      ;
04          000000 .DUSR    PTUSS=  0           ;TUS START
05          000001 .DUSR    PTUSZ=  PTUSS+1     ;TUS SIZE
06          000002 .DUSR    PTSHS=  PTUSZ+1     ;TSH START
07          000003 .DUSR    PTSHZ=  PTSHS+1     ;TSH SIZE
08          000004 .DUSR    PEBPT=  PTSHZ+1     ;4 WORDS OF ECS POINTERS
09
10          000010 .DUSR    PRLN=   PEBPT+4     ;SIZE OF RESIDENT EXTENDER
```

```
!0023  PARS
01                ;
02                ; SWAPPABLE PROCESS TABLE ENTENSION OFFSETS
03                ;
0
                  ;*** NOTE *** PSQCT IS REFERENCED INDIRECTLY- MUST BE 0
          000000 .DUSR    PSQCT=   0         ;ACTIVE SYSTEM PATH COUNT
07        000001 .DUSR    PDCCB=   PSQCT+1   ;DEFAULT DIRECTORY CCB
08        000002 .DUSR    PWCCB=   PDCCB+1   ;WORKING DIRECTORY CCB
09        000003 .DUSR    PBLMX=   PWCCB+1   ;MAX # BLKS
10        000004 .DUSR    PSQMX=   PBLMX+1   ;MAX ACTIVE PATHS
11        000005 .DUSR    PSCCB=   PSQMX+1   ;SAVE FILE CCB (FOR IMPLICIT SHARING)
12        000006 .DUSR    PCTSK=   PSCCB+1   ;CURRENT TCB IN CONTEXT
13        000007 .DUSR    PSWD=    PCTSK+1   ;TCB LINK WD (AWAITING STKS)
14        000010 .DUSR    PSLCN=   PSWD+1    ;NUMBER OF SUB-SLICES IN TIME SLICE
15        000011 .DUSR    PSL=     PSLCN+1   ;SUB-SLICE RESIDUE
16        000012 .DUSR    PNOT8=   PSL+1     ; *** NO LONGER USED
17        000013 .DUSR    PDFR=    PNOT8+1   ;DELAY CHAIN START
18        000014 .DUSR    PSIOC=   PDFR+1    ;# OF OUTSTANDING I/O'S TO MCA'S/
19                                           ; OR LPBS
20        000015 .DUSR    PNOT6=   PSIOC+1   ;NOT USED
21        000016 .DUSR    PMST=    PNOT6+1   ;MAP STATUS
22        000017 .DUSR    PRUNH=   PMST+1    ;ELAPSED RUN TIME IN TICS
23        000020 .DUSR    PRUNL=   PRUNH+1   ;ELAPSED RUN LOW WD
24        000021 .DUSR    PTODH=   PRUNL+1   ;CREATE TOD HIGH
25        000022 .DUSR    PTODL=   PTODH+1   ;CREATE TOD LOW
26        000023 .DUSR    PMXPR=   PTODL+1   ;MAX PRI CHANGE CAN REACH
27        000024 .DUSR    PSRCH=   PMXPR+1   ;SEARCH LIST CCBS
28        000034 .DUSR    PNOT7=   PSRCH+8.  ; NOT USED
29        000035 .DUSR    PTODD=   PNOT7+1   ;CONNECT DAY
          000036 .DUSR    PIOTH=   PTODD+1   ;# I/O BLKS HI
          000037 .DUSR    PIOTL=   PIOTH+1   ;# I/O BLKS LOW
32        000040 .DUSR    PUNM=    PIOTL+1   ;USER NAME
33        000050 .DUSR    PUNEN=   PUNM+8.   ;END OF USER NAME
34        000010 .DUSR    PUNLE=   PUNEN-PUNM ;LENGTH OF USER NAME (LAST WORD IS ACL
35        000051 .DUSR    PHPTB=   PUNEN+1   ;HISTOGRAM TARGET PTBL
36        000052 .DUSR    PHTCB=   PHPTB+1   ;HISTOGRAM TCB
37        000053 .DUSR    PHWDS=   PHTCB+1   ;HISTOGRAM WD GROUPING
38        000054 .DUSR    PHST=    PHWDS+1   ;START OF AREA
39                ; NOTE- PHEND MUST FOLLOW FOR CLM USAGE
40        000055 .DUSR    PHEND=   PHST+1    ;END OF AREA
41        000056 .DUSR    PHADR=   PHEND+1   ;START OF HISTOGRAM ARRAY
42        000057 .DUSR    PHASZ=   PHADR+1   ;ARRAY MAP SIZE
43        000060 .DUSR    PKCTH=   PHASZ+1   ;PAGE-MILLS HI
44        000061 .DUSR    PKCTL=   PKCTH+1   ;PAGE-MILLS LOW
45        000062 .DUSR    PKTIH=   PKCTL+1   ;LAST TIME INTEGRAL HI
46        000063 .DUSR    PKTIL=   PKTIH+1   ;LAST TIME INTEGRAL LOW
47        000064 .DUSR    PRES=    PKTIL+1   ;ADDR OF ROOT PTBL
48        000065 .DUSR    PCLCG=   PRES+1    ;CALL CHARGE IN CLOCK TICKS
49        000066 .DUSR    PRTCB=   PCLCG+1   ;TCB WHO ISSUED ?TERM
50        000067 .DUSR    PRTCD=   PRTCB+1   ;TERM ERROR CODE
51        000070 .DUSR    PRTMS=   PRTCD+1   ;ADDR RTN IPC HDR IN USER SPACE
52        000071 .DUSR    PCEXV=   PRTMS+1   ;CURRENT TASK WITH EXTENDED VARIABLES
53        000072 .DUSR    PCFPU=   PCEXV+1   ;CURRENT PTBL FPU SAVE TASK
54        000073 .DUSR    PFPSV=   PCFPU+1   ;18. WORD FPU SAVE AREA
          000114 .DUSR    PFPEN=   PFPSV+17. ;LAST FPU SAVE AREA LOCATION
          000115 .DUSR    PMXCH=   PFPEN+1   ;MAX CPU TIME ALLOCATED (HIGH)
          000116 .DUSR    PMXCL=   PMXCH+1   ;MAX CPU TIME ALLOCATED (LOW)
58        000117 .DUSR    PGRM=    PMXCL+1   ;2 WD GHOST REMAP FLAGS
59        000120 .DUSR    PGRM2=   PGRM+1
60        000121 .DUSR    PSWBH=   PGRM2+1   ;SWAP DISK ADDR HIGH
```

```
01          000122 .DUSR   PSWBL=   PSWBH+1 ;SWAP DISK ADDR LOW
02          000123 .DUSR   PSWSZ=   PSWBL+1 ;SWAP AREA SIZE IN 1K PAGES
03          000124 .DUSR   PCBPT=   PSWSZ+1 ;4 WORDS OF CCB POINTERS
04
05                 ; THIS IS THE END OF THE COMMON SWAPPABLE PTBL AREA
06          000130 .DUSR   PSCEN=   PCBPT+4 ;DEFINES LAST COMMON LOCATION
07
08
09
10
11
12
13
14
15
16
17
18                 ; DEFINE UNIQUE PARAMETERS FOR THE DEMAND PAGED SYSTEM
19          000000         .IFN    NEWDP
20
21                 .DUSR   PFTOT=  PSCEN+1         ;TOTAL TIME SINCE LAST FAULT
22                 .DUSR   PLFTM=  PFTOT+1         ;PIT DOA LAST TIME PROC RAN
23                 .DUSR   PSP=    PLFTM+1         ;TEMP FOR STACK POINTER ON PEND
24                 ; DEFINE 2 TWO WORD AREAS FOR THE PRIMARY AND GHOST FAULT BITS
25                 .DUSR   PFBT=   PSP+1           ;PRIMARY BITS
26                 .DUSR   PGFBT=  PFBT+2          ;GHOST BITS
27                 .DUSR   PVLD=   PGFBT+2         ;VALIDITY BITS
28
29                 .DUSR   PESZ=   PVLD+2          ;SIZE OF EXTENSION
30
31                 ; NOW DEFINE THE PER-PROCESS FAULT BLOCK
32
33                 .DUSR   PFBK=   PVLD+2          ;FAULT BLOCK
34                 .DUSR   PFBEN=  PFBK+FBLGN-1    ;END LOCATION OF FAULT BLOCK
35
36                 ; REDEFINE SOME OFFSETS WITHIN THE FAULT BLOCK FOR USE BY THE
37                 ; SOFT FAULT ROUTINES AND THE STATE SAVE FOR A PROCESS IN
38                 ; SUPERVISOR (NO CURRENT TCB) MODE.  ALSO USED TO TEMPORARILY
39                 ; SAVE THE PROCESS STATE WHEN A MAP VIOLATION OCCURS.
40
41                 .DUSR   PMAC0=  PFBK+FLAC0      ;SUPERVISOR STATE AC0
42                 .DUSR   PMAC1=  PFBK+FLAC1      ;SUPERVISOR STATE AC1
43                 .DUSR   PMAC2=  PFBK+FLAC2      ;SUPERVISOR STATE AC2
44                 .DUSR   PMAC3=  PFBK+FLAC3      ;SUPERVISOR STATE AC3
45                 .DUSR   PMPC=   PFBK+FLGR3      ;SUPERVISOR STATE PC
46
47                         .ENDC
48
49                 ; DEFINE THE UNIQUE PARAMETERS FOR THE E-200 SYSTEM
50          000001         .IFE    NEWDP
51
52          000131 .DUSR   PMAC0=  PSCEN+1 ; SUPERVISOR STATE AC0
53          000132 .DUSR   PMAC1=  PMAC0+1 ; SUPERVISOR STATE AC1
54          000133 .DUSR   PMAC2=  PMAC1+1 ; SUPERVISOR STATE AC2
55          000134 .DUSR   PMAC3=  PMAC2+1 ; SUPERVISOR STATE AC2
56          000135 .DUSR   PMPC=   PMAC3+1 ; SUPERVISOR STATE PC
57
58          000136 .DUSR   PESZ=   PMPC+1  ; SIZE OF E200 EXTENDER
59
60                         .ENDC
```

```
0025  PARS
01                ; PROCESS TABLE BIT PARAMETERS
02
03                ; STATUS BITS
04
                  ; STATUS BITS HAVE COMMON MEANING FOR BOTH CB'S AND PTBLS
                  ; SO DON'T REDEFINE THEM....
07
08
09        100000 .DUSR    PSRDY=   160              ;NOT READY TO RUN
10        040000 .DUSR    PSETR=   161              ;DON'T ENTER
11
12                ; FOLLOWING STATUS BITS USED IN SSCHED AS A DSPA VALUE
13                ; HIGHEST PRI FUNCTION HAS LOWEST BIT ASSIGNMENT
14        020000 .DUSR    PSPND=   162              ;PENDED ON SOME EVENT
15        010000 .DUSR    PSEW=    163              ;SCHED ACTION
16        004000 .DUSR    PSBRK=   164              ;OP INTERRUPT
17        002000 .DUSR    PSUNP=   165              ;UNPEND THIS PTBL
18        001000 .DUSR    PSBAG=   166              ;SWAP OUT PROC
19        000400 .DUSR    PSBLK=   167              ;BLK PROC
20        000200 .DUSR    PSDP=    168              ;START UP DAEMON
21        000100 .DUSR    PSMWT=   169              ;WAIT FOR MEMORY KEY TO CHANGE
22        000040 .DUSR    PSTSU=   1610             ;TIME SLICE IS UP
23        000020 .DUSR    PSPOP=   1611             ;RTN TO USER VIA A DPOP
24        000004 .DUSR    PSSLN=   1613             ;SYNC LINE I/O COMPLETION BIT
25
26                ; RUN BIT IS NOT USED VIA DSPA
27        000010 .DUSR    PSRUN=   1612             ;RUNNING
28
29
                  ; BIT ADDR
          000100 .DUSR    BPSRY=   PSTAT*1611+0
          000101 .DUSR    BPSEN=   PSTAT*1611+1
33        000102 .DUSR    BPSPN=   PSTAT*1611+2
34        000103 .DUSR    BPSEW=   PSTAT*1611+3.
35        000104 .DUSR    BPSBR=   PSTAT*1611+4.
36        000105 .DUSR    BPSPU=   PSTAT*1611+5.
37        000106 .DUSR    BPSBG=   PSTAT*1611+6.
38        000107 .DUSR    BPSBL=   PSTAT*1611+7.
39        000110 .DUSR    BPSDP=   PSTAT*1611+8.
40        000111 .DUSR    BPSMW=   PSTAT*1611+9.
41        000112 .DUSR    BPSTU=   PSTAT*1611+10.
42        000113 .DUSR    BPSPOP=  PSTAT*1611+11.
43        000114 .DUSR    BPSRN=   PSTAT*1611+12.
44
45                ; PFLAG BITS
46        100000 .DUSR    PFNIN=   180              ;NO OP INTERRUPTS
47        040000 .DUSR    PFPRE=   181              ;PREEMPTIVE RESIDENT
48        020000 .DUSR    PFDEB=   182              ;DEB ENTRY
49        010000 .DUSR    PFFIR=   183              ;FIRST EXECUTION AND LOAD
50        004000 .DUSR    PFELG=   184              ;PROCESS IS ELIG (IN CORE)
51        002000 .DUSR    PFTRP=   185              ;TRAP BIT
52        001000 .DUSR    PFINT=   186              ;CONTROL A FLAG
53        000400 .DUSR    PFBRP=   187              ;BREAK REQUEST
54        000200 .DUSR    PFTRM=   188              ;TERM PROC
          000100 .DUSR    PFMBL=   189              ;PROC CAN ONLY BE EXPLICITLY UNBL
          000040 .DUSR    PFUBL=   1610             ;REQUEST TO UNBLOCK PROCESS
          000020 .DUSR    PFRSH=   1611             ;RESCHED FLAG
58                ;                UNUSED
59        000004 .DUSR    PFSFT=   1613             ;SWAP PROC WON'T FIT NOW
60        000002 .DUSR    PFSWP=   1614             ;SWAPPING TASK
```

```
01                                      1B15            ;WAITING FOR SON TERMINATION
02
03              ;BIT ADDR
04
05      000240 .DUSR    BPFNI=  PFLAG*1B11+0
06      000241 .DUSR    BPFPR=  PFLAG*1B11+1
07      000242 .DUSR    BPFDB=  PFLAG*1B11+2
08      000243 .DUSR    BPFIR=  PFLAG*1B11+3
09      000244 .DUSR    BPFEL=  PFLAG*1B11+4
10      000245 .DUSR    BPFTR=  PFLAG*1B11+5
11      000246 .DUSR    BPFIN=  PFLAG*1B11+6
12      000247 .DUSR    BPFBR=  PFLAG*1B11+7
13      000250 .DUSR    BPFTM=  PFLAG*1B11+8.
14      000251 .DUSR    BPFMB=  PFLAG*1B11+9.
15      000252 .DUSR    BPFUB=  PFLAG*1B11+10.
16      000253 .DUSR    BPFRS=  PFLAG*1B11+11.
17              ;               UNUSED
18      000255 .DUSR    BPFSF=  PFLAG*1B11+13.
19      000256 .DUSR    BPFSW=  PFLAG*1B11+14.
20      000257 .DUSR    BPFEB=  PFLAG*1B11+15.
21
```

```
!0027  PARS
01                ; PFLG2 BITS
02
03          100000 .DUSR    PFSP=    1B0              ;SWAP OUT/IN IN PROGRESS
            040000 .DUSR    PFSU=    1B1              ;UNPEND SOMEONE WAITING FOR SWAP
            020000 .DUSR    PFCMQ=   1B2              ;PROCESS IS ENQUEUED TO CM
            010000 .DUSR    PFDUP=   1B3              ;UNPEND TCB AT HEAD OF DELAY CHN
07          004000 .DUSR    PFIPL=   1B4              ;IPC ORR CHAIN & SPOOL FILE LOCK
08          002000 .DUSR    PFIP2=   1B5              ;TASK WAITING ON PFIPL UNLOCK
09          001000 .DUSR    PFWTO=   1B6              ; BLK TIME OUT
10                ;                  UNUSED
11          000200 .DUSR    PFWSL=   1B8.             ;WAITING ON .SGNL
12          000100 .DUSR    PFSWO=   1B9.             ;PROCESS IS BEING SWAPPED OUT
13          000040 .DUSR    PFBWT=   1B10.            ;WAIT FOR TIME OUT
14          000020 .DUSR    PFATL=   1B11.            ;FATAL PROC TERM
15          000010 .DUSR    PFSUP=   1B12.            ;SUPERUSER MODE
16          000004 .DUSR    PFASG=   1B13.            ;MEM WAIT BIT
17          000002 .DUSR    PFSHC=   1B14.            ;SHARED AREA CHANGED
18          000001 .DUSR    PFQSC=   1B15.            ;INIHBIT SCAN OF BACKED UP TCB RE
19
20
21                ; BIT ADDR FOR PFLG2
22
23          000260 .DUSR    BPFSP=   PFLG2*1B11+0
24          000261 .DUSR    BPFSU=   PFLG2*1B11+1
25          000262 .DUSR    BPFCM=   PFLG2*16.+2
26          000263 .DUSR    BPFDU=   PFLG2*1B11+3.
27          000264 .DUSR    BPFIL=   PFLG2*16.+4.
28          000265 .DUSR    BPFI2=   PFLG2*16.+5.
29          000266 .DUSR    BPFTO=   PFLG2*16.+6.
                  ;                  UNUSED
            000270 .DUSR    BPFWS=   PFLG2*16.+8.
            000271 .DUSR    BPFSO=   PFLG2*16.+9.
33          000272 .DUSR    BPFBW=   PFLG2*16.+10.
34          000273 .DUSR    BPFTL=   PFLG2*16.+11.
35          000274 .DUSR    BPSUP=   PFLG2*16.+12.    ;SUPERUSER MODE
36          000275 .DUSR    BPFAS=   PFLG2*16.+13.
37          000276 .DUSR    BPFSC=   PFLG2*16.+14.    ;SHARED AREA CHANGED CONTENTS
38          000277 .DUSR    BPFQS=   PFLG2*16.+15.    ;INHIBIT TCB REQ SCAN
```

```
01
02                ; PFLG3 BITS
03
04        100000 .DUSR    PFTSE=   160              ;AT LEAST  1 TIME SLICE ENDED
05        040000 .DUSR    PFUBD=   161              ;UNBLK DAD ON RTN
06        020000 .DUSR    PFPTM=   162              ;PROCESSING A TERM
07        010000 .DUSR    PFPCN=   163              ;PROCESSING A CHAIN
08        004000 .DUSR    PFCIE=   164              ;HAS CREATED AN IPC TYPE ENTRY
09        002000 .DUSR    PFIWC=   165              ;INTERRUPT WITHOUT USER CONTROL A
10        001000 .DUSR    PFIRS=   166              ;INT WORLD INTERRUPTED TASK
11        000400 .DUSR    PFRSL=   167              ;FIRST SLICE HAS NOT OCCURRED
12        000200 .DUSR    PFSTM=   168              ;SELF TERM OCCURED
13        000100 .DUSR    PFDIN=   169              ;DELAYED OP INT FLAG
14        000040 .DUSR    PFCFL=   1B10.            ;CCB'S ARE FAULTED OUT
15        000020 .DUSR    PFCUS=   1B11.            ;CCB'S WERE REFERENCED IN LAST SL
16        000010 .DUSR    PFCLD=   1B12.            ;LOAD IN CCB'S (DAEMON BIT)
17        000004 .DUSR    PFPCH=   1B13.            ;HOLD ON >1 PARALLEL CALL
18        000002 .DUSR    PFRBI=   1B14.            ;PROC ASSOCIATED WITH INFOS
19        000001 .DUSR    PFIDF=   1B15             ;RESIDENCY NEEDED (LPB&MCA)
20
21
22                ; BIT ADDR FOR PFLG3
23
24        000300 .DUSR    BPFTE=   PFLG3*16.+0
25        000301 .DUSR    BPFUD=   PFLG3*16.+1
26        000302 .DUSR    BPFFT=   PFLG3*16.+2
27        000303 .DUSR    BPFPC=   PFLG3*16.+3
28        000304 .DUSR    BPFCI=   PFLG3*16.+4      ;HAS CREATED AN IPC ENTRY
29        000305 .DUSR    BPFIW=   PFLG3*16.+5
30        000306 .DUSR    BPFIS=   PFLG3*16.+6
31        000307 .DUSR    BPFSL=   PFLG3*16.+7
32        000310 .DUSR    BPFST=   PFLG3*16.+8.
33        000311 .DUSR    BPFDI=   PFLG3*16.+9.
34        000312 .DUSR    BPFCF=   PFLG3*16.+10.    ;CCB'S ARE FAULTED
35        000313 .DUSR    BPFCU=   PFLG3*16.+11.    ;CCB'S WERE REFERENCED
36        000314 .DUSR    BPFCL=   PFLG3*16.+12.    ;CCB LOAD REQ
37        000315 .DUSR    BPFPH=   PFLG3*16.+13.    ;PARALLEL CALL HOLD
38        000316 .DUSR    BPFRI=   PFLG3*16.+14.    ;INFOS ASSOCIATED
39        000317 .DUSR    BPFID=   PFLG3*16.+15.    ;RESIDENCY NEEDED
```

```
!0031  PARS
01                ;
02                ;  SYSTEM TASK CONTROL BLOCK
03
04
                  ; IN ORDER TO FORM A MORE PERFECT UNION BETWEEN THE SYSTEMS,
                  ;  THE CONTROL BLOCK WILL CONTAIN A REDEFINED AREA FOR EACH
07                ; SYSTEM THAT CONTAINS THE PARAMS TOTALLY UNIQUE TO EACH SYSTEM.
08                ; THIS MEANS THAT ONLY MODULES THAT ARE CONDITIONAL MUST BE
09                ; CONDITIONALLY ASSEMBLED.
10
11
12        000000 .DUSR    CATCB=   0                ;USER TCB ADDR
13        000001 .DUSR    CSTK=    CATCB+1          ;FP
14        000002 .DUSR    CCCB=    CSTK+1           ;CCB ADDR OR 180+SLOT(LB),CCB#(RB
15        000003 .DUSR    CMQWD=   CCCB+1           ;ENQUE WORD IF CORE MANAGER
16
17                ; COMMON ENTRIES WITH PROCESS TABLE
18                ;       PSTAT=   4
19                ;       PLNK=    5
20                ;       PBLNK=   6
21                ;       PNGF=    7
22                ;       PPC=     10
23                ;                PKEY=    11      ;UNPEND KEY
24
25
26        000011 .DUSR    CKEY=    PKEY             ;UNPEND KEY
27        000012 .DUSR    CCRSG=   CKEY+1   ;OVERLAY
28        000013 .DUSR    CSTKC=   CCRSG+1  ;STACK BASE
29        000014 .DUSR    CTEMP=   CSTKC+1  ;UTILITY
          000015 .DUSR    CPTAD=   CTEMP+1  ;PROC TABLE ADDR
          000016 .DUSR    CERWD=   CPTAD+1
                  ; DEFINE MAP SLOT WORDS FOR SLOTS 60000-62000
33        000017 .DUSR    CWN60=   CERWD+1  ; 60000 LMP WORD
34        000020 .DUSR    CWN62=   CWN60+1  ; 62000 LMP WORD
35                ; DEFINE MAP SLOT WORDS FOR SLOTS 70000-76000
36        000021 .DUSR    CWN70=   CWN62+1  ;LMP FOR ADDR 70000
37        000022 .DUSR    CWN72=   CWN70+1  ;LMP FOR ADDR 72000
38        000023 .DUSR    CWN74=   CWN72+1  ;LMP FOR ADDR 74000
39        000024 .DUSR    CWN76=   CWN74+1  ;LMP FOR ADDR 76000
40
41        000006 .DUSR    CLSZ=    CWN76-CWN60+1          ;LMP SIZE
42        000025 .DUSR    CBLGT=   CWN76+1
43
44
45
```

```
!0052  PARS
01              ;PHYSICAL MEMORY USAGE DESCRIPTORS
02
03              ; SHARED MEMORY DESCRIPTOR
04              ;
05              ; CANCH => LRU LIST OF CANDIDATES FOR REUSE.
06              ;          CHAINED THROUGH CMFLK.
07              ; OR
08              ; CHAINED FROM AN FCB TO OTHER SHARED MEMORY DESCRIPTORS
09              ;          THROUGH CFFLK.
10
11      000000 .DUSR    CFFLK=  0       ;FORWARD FCB LINK
12      000001 .DUSR    CFBLK=  CFFLK+1 ;BACKWARD FCB LINK
13      000002 .DUSR    CMMAP=  CFBLK+1 ;PHY. PAGE ASSOCIATED WITH HDR
14              ; THE NEXT FIVE WORDS ARE BLMED AND MUST BE CONTIG
15      000003 .DUSR    CMLAH=  CMMAP+1 ; LOGICAL ADDR HI BYTE
16      000004 .DUSR    CMLAL=  CMLAH+1 ;LOGICAL ADDR LOW
17      000005 .DUSR    CMFCB=  CMLAL+1 ;PARENT FCB ADR (VIRTUAL)
18      000006 .DUSR    CFBLH=  CMFCB+1 ;BLK # HIGH
19      000007 .DUSR    CFBLL=  CFBLH+1 ;BLK# LOW
20      000010 .DUSR    CFHFL=  CFBLL+1 ; HASH FORWARD LINK
21      000011 .DUSR    CFHBL=  CFHFL+1 ; HASH BACKWARD LINK
22
23      000012 .DUSR    CFLEN=  CFHBL-CFFLK+1   ;LENGTH OF SHARED PAGE HEADER
24
25              ; DESCRIPTOR FOR MEMORY USED FOR OVERLAYS XOR FCBS.
26              ;
27              ; OVMCH => CHAIN OF DESCRIPTORS USED TO DESCRIBE OVERLAYS.
28              ; OVTAB+OVERLAY NUMBER => OVERLAY DESCRIPTOR IF
29              ;       OVERLAY IS IN MEMORY.
30              ; FCBCH => CHAIN OF DESCRIPTORS USED TO DESCRIBE BLOCKS
31              ;          CONTAINING FCBS.
32
33      000000 .DUSR    CMSFL=  0       ;FORWARD LINK
34      000001 .DUSR    CMSBL=  1       ;BACK LINK
35      000002 .DUSR    CMOVB=  2       ;PHYSICAL BLK #
36      000003 .DUSR    CMOV1=  3       ;OVLY: OVLY # OR -1 IF NOT IN USE
37                                      ;FCB: WORD 1 OF FCB BIT MAP
38      000004 .DUSR    CMAD1=  4       ;OVLY: BASE ADDR (74000 OR 75000)
39                                      ;FCB: WORD 2 OF FCB BIT MAP
40              ;
41              ; THE FOLLOWING ARE NOT USED FOR FCB BLOCK USAGE DESCRIPTORS.
42              ;
43      000005 .DUSR    CMAD2=  5       ;ADDR OF OTHER HDR BUILT OUT OF 16. WDS
44      000006 .DUSR    CMOUC=  6       ;USE COUNT
```

```
!0033  PARS
01                ; MEMORY MAP TABLE ENTRIES.
02                ;
03                ; ENTRY SIZE ASSUMED TO BE EXACTLY 4 WORDS SO FAST MULTIPLY CAN
04                ; CAN BE USED. MEMORY MAP BEGINS AT CBASE.
                  ; ONE 4 WORD ENTRY PER 1K PHYSICAL MEMORY.
                  ; ENTIRE 4 WORD ENTRY CONTAINS ZEROES IF USED FOR NON-OVERLAY
07                ;       SYSTEM CODE.
08                ; FMCHN => CHAIN OF FREE PHYSICAL MEMORY BLOCKS.
09
10
11                ; DEFINE CORE MAP ENTRY FOR THE E200 (NON-DP)
12
13        000001          .IFE    DP
14        000000 .DUSR    CMPFL=  0       ;FORWARD LINK IF ON FREE CHAIN.
15                                        ;OFFSET MUST BE 0.
16        000001 .DUSR    CMPBL=  1       ;BACK LINK-NOT USED
17        000002 .DUSR    CMPST=  2       ;STATUS OF THIS PAGE
18                                        ;SEE STATUS BIT DEFINITIONS
19                                        ;USE COUNT IS IN THE RIGHT BYTE
20        000002 .DUSR    CMPUC=  CMPST   ;REDEFINE FOR USE COUNT
21        000003 .DUSR    CMPPT=  3       ;CURRENT USAGE DESCRIPTION
22                                        ;PHYS. BLK. # IF FREE
23                                        ;SHARED MEMORY DESC. IF SHARED
24                                        ;FCB USAGE DESC. IF USED FOR FCBS.
25                                        ;OVERLAY USAGE DESC. IF USED FOR OVERLAYS
26                                        ;PROCESS TABLE ADDRESS IF GIVEN TO USER
27                                        ;-1 IF ALLOCATED BY GSMEM IN SYSTEM
28                        .ENDC
29
                  ; DEFINE THE CORE MAP ENTRY FOR THE E500 (DP)
          000000          .IFN    DP
33                .DUSR   CMPFL=  0       ;FORWARD LINK IF ON FREE CHAIN.
34                                        ;OFFSET MUST BE 0.
35                .DUSR   CMPBL=  1       ;BACK LINK
36                .DUSR   CMPST=  2       ;STATUS OF THIS PAGE (SEE BELOW)
37                .DUSR   CMPUC=  CMPST   ;USE COUNT IS IN RIGHT BYTE OF STATUS
38                .DUSR   CMPPT=  3       ;CURRENT USAGE DESCRIPTION
39                                        ;PHYS. BLK. # IF FREE
40                                        ;SHARED MEMORY DESC. IF SHARED
41                                        ;PID+SLOT IN MAP IF AN UNSHARED
42                                        ; USER PAGE OR A CONTROL PAGE
43
44                                        ;-1 IF USED BY AOS
45
46                        .ENDC
47
48
49                ; DEFINE THE STATUS BITS FOR A PAGE OF MEMORY
50                ; THE STATUS IS IN THE LEFT BYTE OF CMPST
51                ; THE PAGE'S USE COUNT IS IN THE RIGHT BYTE OF CMPST
52
53        100000 .DUSR    CMIOP=  1B0     ;I/O IN PROGRESS
54        040000 .DUSR    CMMOD=  1B1     ;PAGE IS MODIFI<ABLE ED>
55        020000 .DUSR    CMSH=   1B2     ;THIS IS A SHARED PAGE
          010000 .DUSR    CMERR=  1B3     ;I/O ERROR DETECTED ON PAGE I/O
          004000 .DUSR    CMAUS=  1B4     ;PAGE IN USE BY AOS
58        002000 .DUSR    CMCPG=  1B5     ;PAGE HAS BEEN ACTUALLY MODIFIED
59        001000 .DUSR    CMRZU=  1B6     ;RELEASE TO LRU LIST WHEN COUNT = 0
60        000400 .DUSR    CMUPI=  1B7     ;CALL UNPEND WHEN I/O COMPLETES
```

```
 0034  PARS
01
02                ; DEFINE BIT POINTERS FOR THE STATUS FLAGS
03
04        000040 .DUSR    BCMIP=  CMPST*16.+0      ;IOP
05        000041 .DUSR    BCMMD=  CMPST*16.+1      ;MOD
06        000042 .DUSR    BCMSH=  CMPST*16.+2      ;SHARED
07        000043 .DUSR    BCMER=  CMPST*16.+3      ;I/O ERROR
08        000044 .DUSR    BCMOS=  CMPST*16.+4      ;AOS PAGE
09        000045 .DUSR    BCMHM=  CMPST*16.+5      ;PAGE HARDWARE MODIFIED
10        000046 .DUSR    BCMRZ=  CMPST*16.+6      ;RELEASE TO LRU ON ZERO USE COUNT
11        000047 .DUSR    BCMUP=  CMPST*16.+7      ;CALL UNPEND WHEN I/O COMPLETES
12
```

```
!0029  PARS
01                ; DEFINE BITS IN PFLG4
02
03          100000 .DUSR    PFSLN=   1B0              ;SYNC LINE DAEMON BP
04          040000 .DUSR    PFDIS=   1B1              ;DISCONNECT OF MODEM OCCURRED
05          020000 .DUSR    PFECD=   1B2              ;AN EXTENDED CONTEXT HAS BEEN DEF
06          010000 .DUSR    PFSPR=   1B3              ;SUPERPROCESS MODE
07          004000 .DUSR    PFMRL=   1B4              ;MAX CPU LIMIT IN USE
08          002000 .DUSR    PFNVT=   1B5              ;PROCESS MUST REMAIN BOUND TO VIR
09          001000 .DUSR    PFBVP=   1B6              ;PROCESS IS BOUND TO VIRTUAL PROC
10          000400 .DUSR    PFOBQ=   1B7              ;PROCESS IS ON BLOCKED QUEUE
11          000200 .DUSR    PFACL=   1B8.             ;USER DEFAULT ACL ENABLED
12          000100 .DUSR    PFSRV=   1B9              ;PROCESS IS A SERVER
13          000040 .DUSR    PFSER=   1B10             ;PROCESS WAITING TO DO SERIAL GSM
14          000020 .DUSR    PFKWB=   1B11             ;PROCESS HAS A TASK DOING ?KWAIT
15          000010 .DUSR    PFKIB=   1B12             ;PROCESS HAS ALL ^C^X DISABLED
16          000004 .DUSR    PFISS=   1B13             ;INT SEQ RECEIVED
17
18
19                ; DEFINE BIT POINTERS FOR PFLG4
20
21          000320 .DUSR    BPSLN=   PFLG4*16.+0      ;SYNC LINE DAEMON BP
22          000321 .DUSR    BPFDS=   PFLG4*16.+1      ;DISCONNECT OF MODEM OCCURRED
23          000322 .DUSR    BPECD=   PFLG4*16.+2      ;AN EXTENDED CONTEXT HAS BEEN DEF
24          000323 .DUSR    BPSPR=   PFLG4*16.+3      ;SUPERPROCESS MODE
25          000324 .DUSR    BPMRL=   PFLG4*16.+4      ;MAX CPU LIMIT IN USE
26          000325 .DUSR    BPNVT=   PFLG4*16.+5      ;PROCESS MUST REMAIN BOUND TO VIR
27          000326 .DUSR    BPBVP=   PFLG4*16.+6      ;PROCESS IS BOUND TO VIRTUAL PROC
28          000327 .DUSR    BPOBQ=   PFLG4*16.+7      ;PROCESS IS ON BLOCKED QUEUE
29          000330 .DUSR    BPACL=   PFLG4*16.+8.     ;USER DEFAULT ACL ENABLED
30          000331 .DUSR    BPSRV=   PFLG4*16.+9.     ;PROCESS IS A SERVER
31          000332 .DUSR    BPSER=   PFLG4*16.+10.    ;PROCESS WAITING TO DO SERIAL GSM
32          000333 .DUSR    BPKWB=   PFLG4*16.+11.    ;PROCESS HAS TASK DOING ?KWAIT
33          000334 .DUSR    BPKIB=   PFLG4*16.+12.    ;PROCESS HAS ALL ^C^X DISABLED
34          000335 .DUSR    BPISS=   PFLG4*16.+13.    ;INT SEQ RECEIVED
```

```
01              ;; DEFINE THE PROCESS PRIV BITS
02
03              ; NOTE- SOME OF THESE ARE ALSO DEFINED IN PARU FOR USER PACKETS
04
05
06      000001 .DUSR    PFPRV=  1B15    ;CAN CREATE AN UNLIMITED NUMB OF PROCE
07      000002 .DUSR    PFPTY=  1B14    ;CAN CREATE PROC OF ANY TYPE
08      000004 .DUSR    PFCBS=  1B13    ;CAN BECOME SUPERUSER
09      000010 .DUSR    PVPR=   1B12    ;CAN CREATE/CHANGE ANY PRI
10      000020 .DUSR    PFPRP=  1B11    ;PERPHERIAL PROC PRIV
11      000040 .DUSR    PFPEX=  1B10    ;CAN CREATE SON AND NOT BLK
12      000100 .DUSR    PFUIP=  1B9     ;CAN SPECIFY A NEW USER NAME ON .PROC
13      000200 .DUSR    PFDVE=  1B8     ;CAN ACCESS DEVICES
14      000400 .DUSR    PFIPC=  1B7     ;CAN USE IPC PRIMITIVES
15      001000 .DUSR    PVINF=  1B6     ;CAN BECOME THE INFOS PROCESS
16      002000 .DUSR    PVSP=   1B5     ;CAN BECOME SUPERPROCESS
17
18
19              ;  BIT POINTERS FOR PRIVILEGE BITS
20
21
22      001077 .DUSR    BPFPV=  PRPRV*16.+15.
23      001076 .DUSR    BPFPT=  PRPRV*16.+14.
24      001075 .DUSR    BPCBS=  PRPRV*16.+13.   ;CAN BECOME SUPERUSER
25      001074 .DUSR    BPVPR=  PRPRV*16.+12.
26      001073 .DUSR    BPFPP=  PRPRV*16.+11.
27      001072 .DUSR    BPFEX=  PRPRV*16.+10.
28      001071 .DUSR    BPFUI=  PRPRV*16.+9.
29      001070 .DUSR    BPFDE=  PRPRV*16.+8.
30      001067 .DUSR    BPFIP=  PRPRV*16.+7.    ;CAN USE IPC PRIMITIVES
31      001066 .DUSR    BPVIF=  PRPRV*16.+6     ;INFOS PROCESS PRIV
32      001065 .DUSR    BPVSP=  PRPRV*16.+5     ;CAN BECOME SUPERPROCESS
```

```
!0037  PARS
01              ; ACS MEMORY REQUEST BITS IN SMFLG
02              ; BITS ARE IN ORDER OF PRIORITY OF ALLOCATION (0 HIGHEST)
03
04      100000 .DUSR    SMMEB=  15B     ;MEM
        040000 .DUSR    SMBFR=  1B1     ;BUF POOL EXPAND
        020000 .DUSR    SMOVB=  1B2     ;OVLY POOL EXPAND
07      010000 .DUSR    SMBSB=  1B3     ;SCAN BLKG
08      004000 .DUSR    SMTOB=  1B4     ;LOOK FOR TIMED OUT PROC
09      002000 .DUSR    SMUER=  1B5     ;UNIT ERROR, SEND MSG TO OPER
10      001000 .DUSR    SMCNB=  1B6     ;SHRINK THE CANDIDATE CHAIN
11
12      000000 .DUSR    BSMCH=  0       ;BIT 0= PROC NEEDS MEM TO COMPLETE .SYS
13      000001 .DUSR    BSMBF=  1       ;BIT 1= BUFFER POOL NEEDS MEM
14      000002 .DUSR    BSMOV=  2       ;BIT 2= OVLY AREA NEEDS MEM
15      000003 .DUSR    BSMBS=  3       ;BIT 3= LOOK FOR UNBLKED PROC
16      000004 .DUSR    BSMTO=  4       ;BIT 4= LOOK FOR TIMED OUT PROC
17      000005 .DUSR    BSMUE=  5       ;BIT 5= UNIT ERROR, SEND MSG TO OPER
18      000006 .DUSR    BSMCN=  6       ;BIT 6= SHRINK CANCH
19
20      000006 .DUSR    BSMLN=  6       ;MAX VALID BIT IN SMFLG
21
22      000001          .IFE    NEWDP
23
24              ;SWAP FILE BLOCK PARAMETERS FOR E/200
25
26      000000 .DUSR    SWPTB=  0       ;SWAPPABLE PTBL EXTENSION
27      000001 .DUSR    SWSDS= SWPTB+1  ;SHARED DIRECTORY START
28      000002 .DUSR    SWSDE= SWSDS+1  ;SHARED DIRECTORY END
29      000003 .DUSR    SWCBS= SWSDE+1  ;CCB IMAGE
        000006 .DUSR    SWCBE= SWCBS+3  ;END OF CCB'S
        000007 .DUSR    SWIMG= SWCBE+1  ;PROCESS IMAGE

33                      .ENDC
34
35      000000          .IFN    NEWDP
36
37              ; SWAP FILE BLOCK PARAMETERS FOR DEMAND PAGING
38
39              .DUSR   SWPTB=  0       ;CONTROL PAGE
40              .DUSR   SWSDS=  SWPTB+0 ;SHARED DIRECTORY START
41              .DUSR   SWSDE=  SWSDS+0 ;NO SHARED DIRECTORY IN DPAGS
42              .DUSR   SWCBS=  SWPTB+4 ;CCB IMAGE START
43              .DUSR   SWCBE=  SWCBS+3 ;CCB IMAGE END
44              .DUSR   SWIMG=  SWCBE+1 ;START OF PROCESS IMAGE
45
46                      .ENDC
47
48              ; SWAP DIRECTORY OFFSETS
49
50      000000 .DUSR    SDLAH= 0        ;LOG ADDR HI 1B0 IF SLOT IS WRITE PROTECT
51      000001 .DUSR    SDLAL= 1        ;LOG ADDR LOW
52      000002 .DUSR    SDFCB= 2        ;FCB V(ADDR) OR 0 IF IDLE OR -1 IF A RE-
53                                      ;MAPPED SLOT (ONLY GHOST SLOTS REMAPPED)
54      000003 .DUSR    SDBLH= 3        ;BLK # HI WD OR PRIMARY SLOT THAT
                                        ;THIS GHOST SLOT IS REMAPPED TO
        000004 .DUSR    SDBLL= 4        ;BLK # LOW OR GHOST SLOT THAT IS
                                        ;REMAPPED

59      000005 .DUSR    SDLGN=  SDBLL+1
60
```

```
 0038  PARS
01
02                ; DEFINE TIME OUT CONTROL BLOCK PARAMETERS
03
04          000000 .DUSR    TMLKF=   0          ;FORWARD LINK
05          000001 .DUSR    TMLKB=   1          ;BACK LINK
06          000002 .DUSR    TMCNT=   2          ;# OF SEC TO WAIT
07          000003 .DUSR    TMMSG=   3          ;CONTENTS OF AC0 UPON ROUTINE ENT
08          000004 .DUSR    TMRTN=   4          ;PROCESSING ROUTINE ADDR
09
10          000004 .DUSR    TMLGT=   TMRTN              ;BLOCK SIZE
```

```
!0039  PARS
01                ; REDEFINE TCB OFFSETS FOR ALL REFERENCES WITHOUT ?XXXX
02
03        000000 .DUSR    TLNK=    ?TLNK    ;LINK (NOTE- MUST BE OFFSET 0 !!!)
          000001 .DUSR    TSTAT=   ?TSTAT   ;STATUS BITS
          000002 .DUSR    TSP=     ?TSP     ;STACK POINTER
          000003 .DUSR    TFP=     ?TFP     ;FRAME POINTER
07        000004 .DUSR    TSL=     ?TSL     ;STACK LIMIT
08        000005 .DUSR    TSO=     ?TSO     ;FAULT HANDLER
09        000006 .DUSR    TAC0=    ?TAC0    ;AC0
10        000007 .DUSR    TAC1=    ?TAC1    ;AC1
11        000010 .DUSR    TAC2=    ?TAC2    ;AC2
12        000011 .DUSR    TAC3=    ?TAC3    ;AC3
13        000012 .DUSR    TPC=     ?TPC     ;PC AND CARRY
14        000013 .DUSP    TUSP=    ?TUSP    ;USP
15        000014 .DUSR    TELN=    ?TELN    ;TCB EXTENTION ADDR
16        000015 .DUSR    TFPSV=   ?TFPS    ;FPU SAVE AREA POINTER
17        000016 .DUSR    TCURD=   ?TCUD    ;CURRENT DESCRIPTOR
18        000017 .DUSR    TSYS=    ?TSYS    ;SYSTEM CALL WORD
19        000020 .DUSR    TIDPR=   ?TIDPR   ;TASK ID  PLUS PRI
20        000021 .DUSR    TSLNK=   ?TSLK    ;SYSTM CALL LINK
21        000022 .DUSR    TKLAD=   ?TKAD    ;KILL ADDR
22        000023 .DUSR    TGEXT=   ?TGEX    ;GHOST EXTENSION ADDRESS
23
24        000024 .DUSR    TLN=?TLN
25        000016 .DUSR    TGXLN=?TGXL
26
27
28                ; DEFINE BITS IN CPU CHARACTERISTICS WORD CPCHR
29
          000017 .DUSR    BCCCM=   15.      ;CPU HAS CMV INSTRUCTION
```

```
!0040  PARS
01                ; CONNECTION MANAGER PARAMETERS
02                ;
03                ;
04
05                ; CONNECTION TABLE DEFINITIONS
06
07        000000          .DUSR   CXLNK=  0       ; RESERVED
08        000001          .DUSR   CXSIZ=  1       ; # ENTRIES
09        000002          .DUSR   CXENT=  2       ; OFFSET OF FIRST ENTRY
10
11                ; CONNECTION TABLE ENTRY DEFINITION
12
13        000000          .DUSR   CXPID=  0       ; PID COUPLET (CUST/SERVER)
14        000001          .DUSR   CXSTS=  1       ; CONNECTION STATUS
15        000002          .DUSR   CXELN=  CXSTS-CXPID+1
16
17                ; CONNECTION STATUS BITS
18
19        100000          .DUSR   CXBMC=  180     ; CONN BROKEN BY CUSTOMER
20        000001          .DUSR   CXBMS=  1815    ; CONN BROKEN BY SERVER
21
22                ; CONNECTION STATUS BIT POINTERS
23
24        000020          .DUSR   CXBPC=  CXSTS*16.+0
25        000037          .DUSR   CXBPS=  CXSTS*16.+15.
26
```

```
01                ; COPYRIGHT (C) DATA GENERAL CORPORATION 1977,1978,1979,1980
02                ; ALL RIGHTS RESERVED.
03                ; LICENSED MATERIAL-PROPERTY OF DATA GENERAL CORPORATION

                          .TITL   PARFS
09
10                ;=======================================;
11                ; AOS REVISION 03 FILE SYSTEM PARAMETERS ;
12                ;=======================================;
13
16                ; HARDWARE CONSTANTS
17
18        000020 .DUSR    SCBPW=  16.       ; BITS PER WORD
19        000400 .DUSR    SCDBS=  256.      ; PHYSICAL DISK BLOCK SIZE (IN WORDS)
20        002000 .DUSR    SCMBS=  1024.     ; PHYSICAL MEMORY BLOCK SIZE (IN WORDS)
21
22                ; FILE SYSTEM LOGICAL CONSTANTS
23
24        000037 .DUSR    SCMNL=  31.       ; MAXIMUM FILE NAME LENGTH IN BYTES
25        000003 .DUSR    SCMIL=  3.        ; MAXIMUM INDEX LEVELS
26        000377 .DUSR    SCMPT=  255.      ; MAXIMUM PATHNAME LENGTH IN BYTES
27        000001 .DUSR    SCDES=  1.        ; DEFAULT DATA ELEMENT SIZE
28        000007 .DUSR    SCDHF=  7.        ; DEFAULT HASH FRAME SIZE (DIRECTORIES)
29        000020 .DUSR    FNSSZ=  (SCMNL/2)+1       ; FILENAME SPACE SIZE
30        000200 .DUSR    PTHSZ=  (SCMPT/2)+1       ; PATHNAME SPACE SIZE
31        000010 .DUSR    DEELS=  8.        ; DIRECTORY ELEMENT ALLOCATION SIZE
32        000010 .DUSR    SCLVL=  8.        ; MAXIMUM DIRECTORY TREE DEPTH
33        000176 .DUSR    SCMBB=  126.      ; MAX NUMBER OF BAD BLOCKS IN LDU
          000002 .DUSR    SCMNI=  2.        ; MIN NUMBER OF IOCB'S (MAX FUNCTION OF

                ; MISCELLANEOUS SYSTEM CONSTANTS
37
38        000210 .DUSR    DFLL=   ?DFLL     ; DEFAULT LINE LENGTH
39
40                ; SPECIAL CHARACTERS
41
42        000025 .DUSR    LDELC=  "U-100    ; LINE DELETE CHARACTER
43        000134 .DUSR    LDELE=  "\        ; LINE DELETE ECHO
44        000003 .DUSR    INTCH=  "C-100    ; INTERRUPT SEQUENCE START CHARACTER
45        000012 .DUSR    NL=     12        ; NEW LINE CHARACTER
46        000004 .DUSR    EOFCH=  "D-100    ; END-OF-FILE CHARACTER
47
48                ; DATA LATE RELATED CONSTANTS
49
50        000144 .DUSR    SCDLC=  100.      ; # OF DATA LATES BEFORE POTENTIAL REPORT
51                                          ; IF YOU CHANGE THIS THEN CHANGE SSCDL IN
52                                          ; MACRO.
53        003410 .DUSR    SCDLT=  (0/2)+30.*(0+60.*1)       ; MAX TIME BETWEEN DATA L
54                        ;       ^           ^     ^
55                        ;     SECONDS     MINS   HOURS
56
57                ; DIB RELATED CONSTANTS
58
          000003 .DUSR    SCREV=  3.        ; DISK FORMAT AND FILE SYSTEM REV NUMBER
          000010 .DUSR    SCMPU=  8.        ; MAXIMUM # OF PUS PER LDU
```

```
 0002  PARFS
01
02        000002 .DUSR   SCBBT=  2.        ; DISK ADDRESS OF BAD BLOCK TABLE
03        000003 .DUSR   SCDIB=  3.        ; DISK ADDRESS OF DIB
04        000010 .DUSR   SCVIS=  8.        ; ADR OF START OF VISIBLE SPACE
05
06
07              ; BAD BLOCK TABLE FORMAT
08
09        000000 .DUSR   BBNBB=0           ;NUM OF BAD BLOCKS ON PU
10        000001 .DUSR   BBRAH=BBNBB+1     ;ADDRESS OF REMAP AREA (HI)
11        000002 .DUSR   BBRAL=BBRAH+1     ;ADDRESS OF REMAP AREA (LO)
12        000003 .DUSR   BBRAS=BBRAL+1     ;REMAP AREA SIZE
13        000004 .DUSR   BBBBD=BBRAS+1     ;START OF BAD BLOCK DESCRIPTORS
14
15
16              ;INTRA-DIRECTORY POINTER
17              ;
18              ; 11 BITS BLOCK NUMBER WITHIN FILE
19              ;  5 BITS ELEMENT ADDRESS IN BLOCK
20              ;        ( = WORD ADDRESS DIVIDED BY 8.)
```

```
!0003  PARFS
01                ;
02                ; DIRECTORY DATA BLOCK PARAMETERS
03                ;
0"
                  ; DIPECTORY DATA ELEMENT TYPE CODES

07        000000 .DUSR    DEFRE=  0         ; FREE ELEMENT (MUST = 0)
08        000001 .DUSR    DEFNB=  1         ; FNB
09        000002 .DUSR    DEFAC=  2         ; FAC
10        000003 .DUSR    DEFIB=  3         ; FIB
11        000004 .DUSR    DEFLB=  4         ; FLB
12        000005 .DUSR    DEFUD=  5         ; FUD
13        000006 .DUSR    DEFUI=  6         ; FUI
14
15                ; DIRECTORY DATA BLOCK FIXED OFFSETS
16
17        000000 .DUSR    DENLB=  0         ; NEXT RELATIVE BLOCK NUMBER
18        000001 .DUSR    DELLB=  DENLB+1 ; LAST RELATIVE BLOCK NUMBER
19
20                ; DIRECTORY DATA ELEMENT HEADER
21
22        000000 .DUSR    DETAS=  0         ; ELEMENT TYPE (LEFT)
23                                          ; SIZE IN WORDS (RIGHT)
24                ;DETAS MUST CONTAIN 0 FOR FREE ELEMENTS
```

```
!0004  PARFS
01                ;
02                ; FILE INFORMATION BLOCK (FIB) PARAMETERS
03                ;
04
05        000001 .DUSR    FINLP=  DETAS+1         ; POINTER TO FIRST FNB (IDF)
06        000002 .DUSR    FIACL=  FINLP+1         ; POINTER TO FAC (IDF)
07        000002 .DUSR    FILBP=  FIACL           ; POINTER TO FLB (LINK ONLY) (
08        000003 .DUSR    FIUID=  FIACL+1         ; UNIQUE ID
09        000004 .DUSR    FITCH=  FIUID+1         ; FILE CREATION TIME (HI)
10        000005 .DUSR    FITCL=  FITCH+1         ; FILE CREATION TIME (LO)
11                ;
12                ; WARNING!!!
13                ;       THE FOLLOWING WORDS FROM "FISTS" TO "FIIDR" COMPRISE
14                ;       THE FUNNY FIB WHICH IS COMMON TO THE FIB, THE DIB
15                ;       AND THE FCB. ITS LENGTH IS "FCOML".
16                ;       ANY CHANGES MADE HERE MUST BE MADE IN THE OTHER
17                ;       TWO PLACES.
18                ;
19        000006 .DUSR    FISTS=  FITCL+1         ; FILE STATUS
20        000007 .DUSR    FITYP=  FISTS+1         ; FILE TYPE (RH) AND FORMAT (LH)
21        000010 .DUSR    FICPS=  FITYP+1         ; FILE CONTROL PARAMETERS
22        000010 .DUSR    FIHFS=  FICPS           ; HASH FRAME SIZE (DIRECTORIES)
23        000010 .DUSR    FIDCU=  FIHFS           ; DEVICE CODE (LEFT),
24                                                ; UNIT NUMBER (RIGHT)
25                                                ; UNIT TYPE ONLY
26        000010 .DUSR    FIHID=  FIHFS           ; HOST ID - NETWORK TYPE FILES
27
28        000011 .DUSR    SFIBL=  FICPS-DETAS+1   ; SHORT FIB LENGTH
29
30                ; FIB EXTENSION FOR DATA FILES AND DIRECTORIES
31
32        000011 .DUSR    FIFW1=  FICPS+1         ; EXTENSION FOR EOF IN FUTURE
33        000012 .DUSR    FIFW2=  FIFW1+1         ;       "       "       "
34        000013 .DUSR    FIEFH=  FIFW2+1         ; LAST LOGICAL BYTE (EOF) (HI)
35        000014 .DUSR    FIEFL=  FIEFH+1         ; LAST LOGICAL BYTE (EOF) (LO)
36        000015 .DUSR    FIDFH=  FIEFL+1         ; DATA ELEMENT SIZE (HI)
37        000016 .DUSR    FIDFL=  FIDFH+1         ; DATA ELEMENT SIZE (LO)
38        000017 .DUSR    FIFAH=  FIDFL+1         ; FIRST LOGICAL ADDRESS (HI)
39        000020 .DUSR    FIFAL=  FIFAH+1         ; FIRST LOGICAL ADDRESS (LO)
40        000021 .DUSR    FIIDX=  FIFAL+1         ; CURRENT INDEX LEVELS (LEFT)
41                                                ; MAXIMUM INDEX LEVELS (RIGHT)
42        000022 .DUSR    FIIDR=  FIIDX+1         ; COUNT OF INFERIOR DIRECTORIES
43        000023 .DUSR    FIFUD=  FIIDR+1         ; POINTER TO FUD
44        000024 .DUSR    FITAH=  FIFUD+1         ; TIME LAST ACCESSED (HI)
45        000025 .DUSR    FITAL=  FITAH+1         ; TIME LAST ACCESSED (LO)
46        000026 .DUSR    FITMH=  FITAL+1         ; TIME LAST MODIFIED (HI)
47        000027 .DUSR    FITML=  FITMH+1         ; TIME LAST MODIFIED (LO)
48        000030 .DUSR    FIFW3=  FITML+1         ; EXTENSION FOR FCB ADDRESS
49        000031 .DUSR    FIFCB=  FIFW3+1         ; VIRTUAL FCB ADDRESS OR ZERO
50
51        000032 .DUSR    FIBLT=  FIFCB-DETAS+1   ; FULL FIB LENGTH
52
53        000015 .DUSR    FCOML=  FIIDR-FISTS+1   ; LENGTH OF COMMON DATA
54                                                ;       BETWEEN FIB AND FCB
55
56                ; FIB EXTENSION FOR CONTROL POINT DIRECTORIES
57
58        000032 .DUSR    FICSH=  FIFCB+1         ; CURRENT SIZE (HIGH)
59        000033 .DUSR    FICSL=  FICSH+1         ; CURRENT SIZE (LOW)
60        000034 .DUSR    FIMSH=  FICSL+1         ; MAX SIZE (HIGH)
```

```
 0005  PARFS
01          000035 .DUSR    FIMSL=  FIMSH+1             ; MAX SIZE (LOW)
02
03          000036 .DUSR    LFIBL=  FIMSL-DETAS+1       ; LONG FIB LENGTH
```

```
:0008  PARFS
01                ; IPC FILE DIFFERENCES
02
03        000013 .DUSR    FIPHI=  FIEFH           ; HOST ID / PID
04        000014 .DUSR    FIPLO=  FIEFL           ; LOCAL PORT NUMBER
05
06                ; MAG TAPE UNIT DIFFERENCES
07
08                ;                                                INITIALIZE TO
09        000013 .DUSR    FILFL=  FIEFH           ; LOG EOT FILE            -1
10        000014 .DUSR    FILBL=  FIEFL           ; LOG EOT BLOCK           -1
11        000015 .DUSR    FIFIL=  FIDFH           ; CURRENT FILE NUMBER     -1
12        000016 .DUSR    FIBLK=  FIDFL           ; CURRENT BLOCK NUMBER     X
13        000017 .DUSR    FIOBL=  FIFAH           ; OLD BLOCK COUNT         -1
14        000020 .DUSR    FIOB2=  FIFAL           ; SECOND OLD BLOCK COUNT   0
15
16                ; STATUS BIT DEFINITIONS AND POINTERS ARE WITH FCB DEFINITION
```

```
!0007  PARFS
01                ;
02                ; FILE NAME BLOCK (FNB) PARAMETERS
03                ;
04
          000001 .DUSR    FNFIB=  DETAS+1          ;FIB POINTER
          000002 .DUSR    FNNAM=  FNFIB+1          ;FILE NAME OFFSET

08        000002 .DUSR    FNBLT=  FNNAM-DETAS      ; FNB HEADER LENGTH
09
10
11
12
13
14                ;
15                ; FILE ACCESS CONTROL BLOCK (FAC) PARAMETERS
16                ;
17
18        000001 .DUSR    FAFIB=  1        ; POINTER TO FIB
19        000002 .DUSR    FAACL=  2        ; ACCESS CONTROL LIST OFFSET
20
21        000002 .DUSR    FACLT=  FAACL-DETAS      ; FAC HEADER LENGTH
22
23
24
25
26
27                ;
28                ; FILE LINK BLOCK (FLB) PARAMETERS
29                ;
30
          000001 .DUSR    FLFIB=  1        ; POINTER TO FIB
          000002 .DUSR    FLLCN=  2        ; LINK DATA OFFSET
33
34        000002 .DUSR    FLBLT=  FLLCN-DETAS      ;FLB HEADER LENGTH
35
36
37
38
39                ;
40                ; FILE USER DATA BLOCK (FUD) PARAMETERS
41                ;
42
43        000001 .DUSR    FUFIB=  1        ;FIB POINTER
44        000002 .DUSR    FUFFL=  2        ;FUD FORWARD LINK
45        000003 .DUSR    FUFBL=  3        ;FUD BACKWARD LINK
46        000004 .DUSR    FUUDA=  4        ;USER DATA OFFSET
47
48        000004 .DUSR    FUDLT=  FUUDA    ;FUD HEADER LENGTH
49
50                ;
51                ; FILE UNIQUE ID BLOCK (FUI) PARAMETERS
52                ;
53
54        000001 .DUSR    FDFIB=  1        ;FIB POINTER
55        000002 .DUSR    FDUID=  2        ;ID DATA OFFSET

          000002 .DUSR    FUILT=  FDUID    ;FUI HEADER LENGTH
```

```
:0006  PARPS
01                      ;
02                      ; ACCESS CONTROL PRIVILEGE PARAMETERS
03                      ;
04
05                      ; ACCESS CONTROL PRIVILEGE BIT POSITIONS
06
07          000017 .DUSR    APEXC=  ?FAEB   ; EXECUTE IS ENABLED
08          000016 .DUSR    APRED=  ?FARB   ; READ IS ENABLED
09          000015 .DUSR    APAPN=  ?FAAB   ; APPEND IS ENABLED
10          000014 .DUSR    APWRT=  ?FAWB   ; WRITE IS ENABLED
11          000013 .DUSR    APOWN=  ?FAOB   ; OWNER ACCESS
12
13                      ; ACCESS CONTROL PRIVILEGE MASKS
14
15          000001 .DUSR    APEMK=  1B(APEXC)       ; EXECUTE
16          000002 .DUSR    APRMK=  1B(APRED)       ; READ
17          000004 .DUSR    APAMK=  1B(APAPN)       ; APPEND
18          000010 .DUSR    APWMK=  1B(APWRT)       ; WRITE
19          000020 .DUSR    APOMK=  1B(APOWN)       ; OWNER
20
21                      ; MASK OF ALL PRIVILEGE BITS
22
23          000037 .DUSR    APMSK=  APEMK+APRMK+APAMK+APWMK+APOMK
```

```
!0009  PARFS
01                ;
02                ; DISK INFORMATION BLOCK (DIB) PARAMETERS
03                ;
04
          000000 .DUSR    IBREV=0           ;DISK FORMAT AND FILE SYSTEM REV NUMBER
          000001 .DUSR    IBTYP=IBREV+1     ;DISK UNIT TYPE
07        000002 .DUSR    IBSTS=IBTYP+1     ;STATUS WORD(PER UNIT FLAGS???)
08        000003 .DUSR    IBIDH=IBSTS+1     ;LDU UNIQUE ID (HIGH)
09        000004 .DUSR    IBIDM=IBIDH+1     ;LDU UNIQUE ID (MIDDLE)
10        000005 .DUSR    IBIDL=IBIDM+1     ;LDU UNIQUE ID (LOW)
11        000006 .DUSR    IBSNP=IBIDL+1     ;SEQUENCE NUMBER OF THIS PU IN LDU
12        000007 .DUSR    IBNPU=IBSNP+1     ;NUMBER OF PUS IN LDU
13        000010 .DUSR    IBNHD=IBNPU+1     ;NUMBER OF HEADS
14        000011 .DUSR    IBNST=IBNHD+1     ;NUMBER OF SECTORS PER TRACK
15        000012 .DUSR    IBNCY=IBNST+1     ;NUMBER OF CYLINDERS
16        000013 .DUSR    IBVIS=IBNCY+1     ;DISK ADDRESS OF START OF VISIBLE SPACE
17        000014 .DUSR    IBNBH=IBVIS+1     ;NUMBER OF VIS DISK BLOCKS (HIGH)
18        000015 .DUSR    IBNBL=IBNBH+1     ;NUMBER OF VIS DISK BLOCKS (LOW)
19        000016 .DUSR    IBBTH=IBNBL+1     ;PHYS ADDR OF BAD BLOCK TABLE (HI)
20        000017 .DUSR    IBBTL=IBBTH+1     ;PHYS ADDR OF BAD BLOCK TABLE (LO)
21        000020 .DUSR    IBUID=IBBTL+1     ;10 WORD UNIQUE ID FOR N.C.
22
23                ; THE FOLLOWING DIB OFFSETS ARE ONLY VALID ON UNIT 1 OF THE LDU
24
25        000032 .DUSR    IBLDF=IBUID+10.   ;LD FLAGS
26        000033 .DUSR    IBNMH=IBLDF+1     ;DISK ADDRESS OF NAME BLOCK (HI)
27        000034 .DUSR    IBNML=IBNMH+1     ;DISK ADDDRESS OF NAME BLOCK (LO)
28        000035 .DUSR    IBACH=IBNML+1     ;DISK ADDRESS OF ACCESS CONTROL BLOCK (HI
29        000036 .DUSR    IBACL=IBACH+1     ;DISK ADDRESS OF ACCESS CONTROL BLOCK (LO
30        000037 .DUSR    IBBAH=IBACL+1     ;DISK ADDRESS OF BITMAP (HI)
          000040 .DUSR    IBBAL=IBBAH+1     ;DISK ADDRESS OF BITMAP (LO)
          000041 .DUSR    IBSBH=IBBAL+1     ;SYSTEM BOOTSTRAP ADDRESS (HI)
33        000042 .DUSR    IBSBL=IBSBH+1     ;SYSTEM BOOTSTRAP ADDRESS (LO)
34        000043 .DUSR    IBSSB=IBSBL+1     ;SIZE OF SYSTEM BOOTSTRAP
35        000044 .DUSR    IBOAH=IBSSB+1     ;ADDRESS OF OVERLAY AREA (HI)
36        000045 .DUSR    IBOAL=IBOAH+1     ;ADDRESS OF OVERLAY AREA (LO)
37        000046 .DUSR    IBOAS=IBOAL+1     ;SIZE OF OVERLAY AREA
38        000047 .DUSR    IBFBP=IBOAS+1     ;IDP TO FIB OF INSTALLED SYSTEM
39
40                ;
41                ; NOTE: THE FOLLOWING FUNNY FIB MUST BE IDENTICAL
42                ;       TO THAT CONTAINED IN THE FIB AND THE FCB
43                ;
44        000050 .DUSR    IBFFB=IBFBP+1     ;FUNNY FIB STARTS HERE (FCOML WORDS)
45
46        000065 .DUSR    IBCSH=IBFFB+FCOML  ;CURRENT SIZE OF LD (HI)
47        000066 .DUSR    IBCSL=IBCSH+1     ;CURRENT SIZE OF LD (LO)
48        000067 .DUSR    IBMSH=IBCSL+1     ;MAX SIZE OF LD (HI)
49        000070 .DUSR    IBMSL=IBMSH+1     ;MAX SIZE OF LD (LO)
50
51        000071 .DUSR    IBLEN=IBMSL+1     ;DIB FIXED LENGTH
52
53
54
55
                ; DIB STATUS WORD MASK DEFINITIONS
                ; VALID ONLY ON FIRST DISK OF LD
59
60        100000 .DUSR    IBSIN=  100             ;LOGICAL DISK INITIALIZED
```

```
0010  PARFS
01        040000 .DUSR   IBSBI=  181                ;SYSBOOT HAS BEEN INSTALLED
```

```
!0011  PAPFS
01                ;
02                ; CHANNEL CONTROL BLOCK PARAMETERS
03                ;
04
                  ;
                  ; THE FIRST FIVE OFFSETS ARE COMMON BETWEEN CHANNEL CONTROL
                  ; BLOCKS AND THE ENQUEUE BLOCKS USED BY DSKIO IN DEMAND PAGED
08                ; ADS AND THEREFORE SHOULD NOT BE CHANGED.
09                ;
10        000000 .DUSR    CBFCB=  0         ; VIRTUAL FCB ADDRESS
11        000001 .DUSR    CBQLK=  1         ; ENQUE LIST LINK WORD
12        000002 .DUSR    CBNBK=  2         ; NUMBER OF BLOCKS TO TRANSFER (RIGHT)
13                                          ; PRIORITY (LEFT)
14        000003 .DUSR    CBPTA=  3         ; PROCESS TABLE ADDRESS
15                                          ;       (180 = 1 => GHOST ADDRESS)
16        000004 .DUSR    CBIAH=  4         ; INDEX BLOCK LOGICAL ADDRESS (HI)
17        000005 .DUSR    CBIAL=  5         ; INDEX BLOCK LOGICAL ADDRESS (LO)
18        000006 .DUSR    CBPCB=  6         ; PARENT CCB POINTER
19        000007 .DUSR    CBFIB=  7         ; FIB POINTER IN SUPERIOR DIRECTORY
20        000010 .DUSR    CBIBN=  10        ; RELATIVE INDEX BLOCK NUMBER (X1X2)
21        000011 .DUSR    CBDBH=  11        ; LAST BLOCK BYTE COUNT - 1 (B0-B8)
22                                          ; RELATIVE DATA BLOCK NUMBER (HI) (B9-B15
23        000012 .DUSR    CBDBL=  12        ; RELATIVE DATA BLOCK NUMBER (LO)
24                                          ; BLOCK #/RETRY COUNT (MTA/MCA)
25        000013 .DUSR    CBUAD=  13        ; USER ADDRESS FOR DIRECT I/O
26        000014 .DUSR    CBSTS=  14        ; STATUS (LEFT 10 BITS)
27                                          ; ACL BITS OR UNIT TYPE IN RIGHT 5 BITS
28        000015 .DUSR    CBTCB=  15        ; USER TCB ADDRESS
29        000016 .DUSR    CBUID=  16        ; FCB NUMBER (UID)
30        000017 .DUSR    CBUPD=  17        ; POST PROCESSING ADDRESS
          000017 .DUSR    CBDEN=  CBUPD     ; DENSITY MODE BITS.

33        000020 .DUSR    CCBLT=  CBUPD-CBFCB+1   ; LENGTH OF CCB
34
35        000011 .DUSR    CBLAH=  CBDBH     ; RETURN DEVICE ADDRESS FOR SHARED BLOCK
36        000015 .DUSR    CBLAL=  CBTCB     ; RETURN DEVICE ADDRESS (LOW)
37
38                ; SOME REDEFINITIONS
39
40        000004 .DUSR    CBVCB=  CBIAH     ; V(CCB) ADDRESS (E500 DISKIO NQ BLOCK)
41        000004 .DUSR    CBMDC=  CBIAH     ; DCT ADDRESS (MCA)
42        000005 .DUSR    CBBLN=  CBIAL     ; BLOCK LENGTH (MAG TAPE)
43        000010 .DUSR    CBBHR=  CBIBN     ; BUFFER HEADER ADDRESS (MAG TAPE)
44        000011 .DUSR    CBQBC=  CBDBH     ; REQUESTED BYTE COUNT (LPB)
45        000011 .DUSR    CBMCL=  CBDBH     ; FILE/LINK # (MTA/MCA)
46        000015 .DUSR    CBUSC=  CBTCB     ; USE COUNT (DIRECTORY)
47
48
49                ;
50                ;  CCB STATUS BIT POSITIONS
51                ;
52
53        000012 .DUSR    CBFNS=  10.       ; FILE NUMBER SET (MAG TAPE)
54        000012 .DUSR    CBIFF=  CBFNS     ; INHIBIT INITIAL FORM FEED (LPB)
55        000012 .DUSR    CBMIF=  CBFNS     ; MESSAGE IN FILE (IPC)
          000011 .DUSR    CBPEB=  9.        ; I/O IS ILLEGAL
          000010 .DUSR    CBEOV=  8.        ; ENABLE VFU LOAD (LPB)
                                            ; OVERRIDE LEOT LOGIC (MAG TAPE)
59                                          ; DO NOT UPDATE EOF (?COPEN)
60        000007 .DUSR    CBC1B=  7.        ; COMMAND BIT 1 (MUST = 7.)
```

```
000006 .DUSR   CBC2B=   6.      ; COMMAND BIT 2 (MUST = 6.)
000005 .DUSR   CBSHB=   5.      ; REQUEST IS SHARED
000004 .DUSR   CBFSH=   4.      ; FILE IS SHARED
000003 .DUSR   CBLKB=   3.      ; CCB LOCK
000002 .DUSR   CBTPB=   2.      ; TASK PENDED ON UNLOCK
000001 .DUSR   CBERB=   1.      ; ERROR OCCURED
000000 .DUSR   CBUNT=   0       ; UNIT TYPE CCB (NOT DISK)


               ;
               ;  CCB STATUS MASKS
               ;

100000 .DUSR   CBUNM=   1B(CBUNT)       ; UNIT TYPE CCB
000200 .DUSR   CBEOR=   1B(CBEOV)       ; ENABLE VFU (LPB)
                                        ; OVERRIDE LEOT LOGIC (MAG TAPE)
                                        ; DO NOT UPDATE EOF (?COPEN)
000040 .DUSR   CBFNM=   1B(CBFNS)       ; FILE NUMBER SET (MAG TAPE)
000040 .DUSR   CBFFM=   1B(CBIFF)       ; INHIBIT INITIAL FORM FEED
000040 .DUSR   CBMES=   1B(CBMIF)       ; MESSAGE IN FILE (IPC)
000100 .DUSR   CBPER=   1B(CBPEB)       ; I/O IS ILLEGAL
000400 .DUSR   CBC1M=   1B(CBC1B)       ; COMMAND BIT 1
001000 .DUSR   CBC2M=   1B(CBC2B)       ; COMMAND BIT 2
002000 .DUSR   CBSHR=   1B(CBSHB)       ; REQUEST IS SHARED
004000 .DUSR   CBFSA=   1B(CBFSH)       ; FILE IS SHARED
010000 .DUSR   CBLCK=   1B(CBLKB)       ; CCB LOCK
020000 .DUSR   CBTPL=   1B(CBTPB)       ; UNPEND ON UNLOCK
040000 .DUSR   CBERM=   1B(CBERB)       ; ERROR


               ;
               ;  CCB STATUS BIT POINTERS
               ;

000310 .DUSR   BCBEO=   CBSTS*SCBPW+CBEOV       ; ENABLE VFU (LPB)
                                                ; OVERRIDE LEOT (MAG TAPE
                                                ; DO NOT UPDATE EOF (?COP
000300 .DUSR   BCBUN=   CBSTS*SCBPW+CBUNT       ; UNIT TYPE CCB
000312 .DUSR   BCBFN=   CBSTS*SCBPW+CBFNS       ; FILE NUMBER SET (MAG TA
000312 .DUSR   BCBFF=   CBSTS*SCBPW+CBIFF       ; INHIBIT INITIAL FF (LPB
000312 .DUSR   BCBMS=   CBSTS*SCBPW+CBMIF       ; MESSAGE IN FILE (IPC)
000311 .DUSR   BCBPE=   CBSTS*SCBPW+CBPEB       ; I/O IS ILLEGAL
000307 .DUSR   BCBC1=   CBSTS*SCBPW+CBC1B
000306 .DUSR   BCBC2=   CBSTS*SCBPW+CBC2B
000305 .DUSR   BCBSH=   CBSTS*SCBPW+CBSHB       ; REQUEST IS SHARED
000304 .DUSR   BCBSA=   CBSTS*SCBPW+CBFSH       ; FILE IS SHARED
000303 .DUSR   BCBLK=   CBSTS*SCBPW+CBLKB       ; CCB LOCK
000302 .DUSR   BCBPL=   CBSTS*SCBPW+CBTPB       ; UNPEND ON UNLOCK
000301 .DUSR   BCBER=   CBSTS*SCBPW+CBERB       ; ERROR


               ;
               ;  NGCCB COMMAND CODES
               ;

000000 .DUSR   CBRED=   0               ;READ
000400 .DUSR   CBWRI=   CBC1M           ;WRITE
001000 .DUSR   CBDEL=   CBC2M           ;DELETE
001400 .DUSR   CBSYB=   CBC1M+CBC2M     ;READ SYSTEM BLOCK
```

```
 0013  PARFS
01                ; DEFINE UNIT TYPES ( IN RIGHT 5 BITS OF CBSTS)
02
03        000000 .DUSR    CUTMT=  0                       ; MAG TAPE
04        000001 .DUSR    CUTMC=  1                       ; MCA
          000002 .DUSR    CUTSL=  2                       ; SYNCH LINE
          000003 .DUSR    CULPB=  3                       ; LINE PRINTER
07        000004 .DUSR    CULPD=  4                       ;LP2 PRINTER
08
09        000037 .DUSR    CUMSK=  37                      ; MASK FOR UNIT TYPE
```

```
10014  PARFS
01                      ;
02                      ; FILE CONTROL BLOCK PARAMETERS
03                      ;
04
05      000000 .DUSR    FBLCB=  0               ; LOGICAL UNIT CONTROL BLOCK (E
06      000001 .DUSR    FBBLP=  FBLCB+1         ; BUFFER LIST POINTER
07                      ;
08                      ; NOTE: FOLLOWING "FCOML" WORDS MUST BE THE SAME AS THE
09                      ;       FUNNY FIB DEFINED IN THE FIB.  ASSEMBLY ERRORS
10                      ;       WILL RESULT OTHERWISE.
11                      ;
12      000002 .DUSR    FBSTS=  FBBLP+1         ; FCB STATUS
13      000003 .DUSR    FBTYP=  FBSTS+1         ; FILE TYPE (RH) AND FORMAT (LH)
14      000004 .DUSR    FBCPS=  FBTYP+1         ; FILE CONTROL PARAMETERS
15      000004 .DUSR    FBHFS=  FBCPS           ; HASH FRAME SIZE (DIRECTORIES)
16      000004 .DUSR    FBDCU=  FBHFS           ; DEV CODE & UNIT NUM (UNIT TYPE F
17      000005 .DUSR    FBFW1=  FBDCU+1         ; EXTENSION FOR EOF IN FUTURE
18      000006 .DUSR    FBFW2=  FBFW1+1         ;       "       "   "   "   "
19      000007 .DUSR    FBEFH=  FBFW2+1         ; LAST BYTE (EOF) ADDRESS (HI)
20      000010 .DUSR    FBEFL=  FBEFH+1         ; LAST BYTE (EOF) ADDRESS (LO)
21      000011 .DUSR    FBDFH=  FBEFL+1         ; DATA ELEMENT SIZE (HI)
22      000012 .DUSR    FBDFL=  FBDFH+1         ; DATA ELEMENT SIZE (LO)
23      000013 .DUSR    FBFAH=  FBDFL+1         ; FIRST LOGICAL ADDRESS (HI)
24      000014 .DUSR    FBFAL=  FBFAH+1         ; FIRST LOGICAL ADDRESS (LO)
25      000015 .DUSR    FBIDX=  FBFAL+1         ; CURRENT INDEX LEVELS (LEFT),
26                                              ; MAXIMUM INDEX LEVELS (RIGHT)
27      000016 .DUSR    FBIDR=  FBIDX+1         ; COUNT OF INFERIOR DIRECTORIES
28                      ;
29                      ; THIS IS THE END OF THE FUNNY FIB.
30                      ; AN ASSEMBLY WILL RESULT IF IT ISN'T "FCOML" WORDS LONG.....
31                      ;
32      000000                  .IFN    FBIDR-FBSTS-FCOML+1
33                      AN ASSEMBLY ERROR ON THESE LINES MEANS THE FUNNY FIB IS MESSED
34                      SEE THE ABOVE COMMENT AND THE FIB DEFINITION FOR MORE INFO.
35                              .ENDC
36
37      000017 .DUSR    FBOPN=  FBIDR+1         ; OPEN COUNT
38      000020 .DUSR    FBUID=  FBOPN+1         ; FCB GLOBAL NUMBER (UID)
39      000021 .DUSR    FBPDP=  FBUID+1         ; PARENT DIRECTORY CCB POINTER
40      000022 .DUSR    FBFIB=  FBPDP+1         ; FIB POINTER
41      000023 .DUSR    FBCMP=  FBFIB+1         ; FCB SHARED BLOCK LIST POINTER
42      000024 .DUSR    FBSCB=  FBCMP+1         ; POINTER TO SYSTEM CCB (OR ZERO)
43      000025 .DUSR    FBLOK=  FBSCB+1         ; RECORD LOCK LIST
44      000026 .DUSR    FBLVL=  FBLOK+1         ; DEPTH OF FILE IN DIRECTORY HIER
45      000027 .DUSR    FANDY=  FBLVL+1         ; USE COUNT
46      000030 .DUSR    FBCPB=  FANDY+1         ; POINTER TO CONTROL POINT BLOCK
47      000031 .DUSR    FBUDB=  FBCPB+1         ; UDB ADDRESS (MTA)
48      000032 .DUSR    FBCLP=  FBUDB+1         ; CACHE LIST POINTER
49
50      000033 .DUSR.   FCBLT=  FBCLP-FBLCB+1   ;LENGTH OF FILE CONTROL BLOCK
```

```
!0015  PARFS
01
02              ; MAG TAPE UNIT DIFFERENCES
03              ;THE NEXT 6 LOCS MUST BE CONTIGIOUS AND ORDERED WITH THE UDB
04        000007 .DUSR    FBLFL=  FBEFH   ; LOGICAL EOT FILE
          000010 .DUSR    FBLBL=  FBEFL   ; LOGICAL EOT BLOCK
          000011 .DUSR    FBFIL=  FBDFH   ; CURRENT FILE
07        000012 .DUSR    FBBLK=  FBDFL   ; CURRENT BLOCK
08        000013 .DUSR    FBOBL=  FBFAH   ; MOST RECENT OLD BLOCK COUNT
09        000014 .DUSR    FBOB2=  FBFAL   ; SECOND MOST RECENT OLD BLK CNT
10
11        000006 .DUSR    FBCOM=FBOB2-FBLFL+1     ;# OF WORDS IN COMMON FCB/UDB
12
13              ;MCA REDEFINITIONS
14              ;
15        000007 .DUSR    FBMCL=  FBEFH           ;INDICATES IF LINK IS SPECIFIED
16                                                ;DYNAMICALLY WITH A R/W OR ONCE
17                                                ;AT OPEN TIME..(TO DETERMINE IF
18                                                ;EOF IS TO BE SENT AT CLOSE TIME)
19        000010 .DUSR    FBMDC=  FBEFL           ;MCA DCT ADDRESS
20
```

```
                ;
                ; BIT MAP FCB REDEFINITIONS.
                ;       THE BIT MAP FCB IS NOT USED LIKE A NORMAL FCB AND
                ;       IS MANIPULATED BY THE WITHDRAW BLOCK AND DEALLOCATE
                ;       BLOCK ROUTINES.
                ;

                ; THESE FIRST 8 LOCATIONS MUST BE INITIALIZED WHEN THE
                ; BIT MAP FCB IS CREATED (MLDUI, DINIT AND XINIT).

000012 BMNB=    FBDFL           ;NUMBER OF BLOCKS MAKING UP BIT MAP
000013 BMFAH=   FBFAH           ;LOGICAL FIRST ADDRESS OF BIT MAP (HI)
000014 BMFAL=   FBFAL           ;                                 (LO)
000005 BMF1H=   FBFW1           ;WDBLK LAST FOUND A FREE BLOCK HERE (HI)
000006 BMF1L=   FBFW2           ;                                   (LO)
000007 BMFNH=   FBEFH           ;WDCBK LAST FOUND "BMPRN" BLKS HERE (HI)
000010 BMFNL=   FBEFL           ;                                   (LO)
000011 BMPRN=   FBDFH           ;# OF BLOCKS THAT WDCBK LAST FOUND

                ; THE FOLLOWING LOCATIONS ARE USED AS TEMPORARIES BY THE BIT
                ; MAP CODE AND NEED NOT BE INITIALIZED.

000003 BMBAH=   FBTYP           ;LOGICAL ADDR OF NEXT BM BLK (HI)
000004 BMBAL=   FBCPS           ;                            (LO)
000016 BMBC=    FBIDR           ;COUNT OF BLKS IN BM YET TO BE SEARCHED
000015 BMWC=    FBIDX           ;CNT OF WORDS IN BLK YET TO BE SEARCHED
000002 BMCNT=   FBSTS           ;# OF CONTIGUOUS BITS YET TO BE
                                ;FOUND OR TURNED ON
000017 BMRC=    FBOPN           ;RELATIVE BLK WITHIN BM OF ROOT BLOCK
000020 BMRD=    FBUID           ;BIT DISPLACEMENT WITHIN ROOT BLOCK OF
                                ;START OF CONTIGUOUS BIT STRING (CALLE
                                ;ROOT BIT DISPLACEMENT). IF -1 - ROOT
                                ;NOT YET LOCATED.
000021 BMT1=    FBPDP           ;TEMPORARY STORAGE 1
000022 BMBLW=   FBFIB           ;# OF BITS LEFT IN THIS WORD
000023 BMTWZ=   FBCMP           ;# OF WORDS OF ZEROS WE MUST FIND
000024 BMWLZ=   FBSCB           ;# OF WORDS OF ZEROS LEFT TO FIND


                ;
                ; CONTROL POINT BLOCK (CPB) PARAMETERS
                ;

000000 .DUSR    CPCPB=  0       ;POINTER TO PARENT CONTROL POINT
                                ;BLOCK (-1 IF NONE)
000001 .DUSR    CPCSH=  CPCPB+1 ;CURRENT SIZE (HIGH)
000002 .DUSR    CPCSL=  CPCSH+1 ;CURRENT SIZE (LOW)
000003 .DUSR    CPMSH=  CPCSL+1 ;MAX SIZE (HIGH)
000004 .DUSR    CPMSL=  CPMSH+1 ;MAX SIZE (LOW)

000005 .DUSR    CPBLT=  CPMSL-CPCPB+1   ;CPB FIXED LENGTH
```

```
!0017  PARFS
01
02                ;
03                ;  FCB STATUS BIT POSITIONS
04                ;

          000000 .DUSR    FBSHB=  0.        ; FILE IS SHARED
07        000001 .DUSR    FBMDB=  1.        ; FILE MODIFIED
08        000002 .DUSR    FBDMB=  2.        ; FILE WAS DUMPED
09        000003 .DUSR    FBLKB=  3.        ; PEND LOCK FOR SHARED I/O REQ
10        000004 .DUSR    FBTPB=  4.        ; TASK PENDED ON PENDLOCK
11        000005 .DUSR    FBPRM=  5.        ; FILE IS PERMANENT
12        000006 .DUSR    FBDLE=  6.        ; DELETE FILE ON LAST CLOSE
13        000007 .DUSR    FBFDB=  7.        ; FILE HAS A USER DATA AREA
14        000010 .DUSR    FBFLB=  8.        ; FLUSH TO FIB
15        000011 .DUSR    FBOEX=  9.        ; FILE EXCLUSIVELY OPENED
16        000012 .DUSR    FBIOP=  10.       ; FILE HAS I/O IN PROGRESS
17                ; UNIVERSAL ACCESS BITS (MUST MATCH CCB AND ACL BITS)
18        000017 .DUSR    FBAEA=  ?FAEB     ; ALL USERS HAVE EXECUTE ACCESS
19        000016 .DUSR    FBARA=  ?FARB     ; ALL USERS HAVE READ ACCESS
20        000015 .DUSR    FBAAA=  ?FAAB     ; ALL USERS HAVE APPEND ACCESS
21        000014 .DUSR    FBAWA=  ?FAWB     ; ALL USERS HAVE WRITE ACCESS
22        000013 .DUSR    FBAOB=  ?FAOB     ; ALL USERS HAVE OWNER ACCESS
23
24
25                ;
26                ;  FCB STATUS MASKS
27                ;
28
29        100000 .DUSR    FBSHR=  1B(FBSHB)         ; FILE IS SHARED
30        040000 .DUSR    FBMOD=  1B(FBMDB)         ; FILE MODIFIED
          020000 .DUSR    FBDMP=  1B(FBDMB)         ; DUMPED
          010000 .DUSR    FBLCK=  1B(FBLKB)         ; PEND LOCK
33        004000 .DUSR    FBTPL=  1B(FBTPB)         ; TASK PENDED ON PENDLOCK
34        002000 .DUSR    FBPRF=  1B(FBPRM)         ; PERMANENT FILE
35        001000 .DUSR    FBDEL=  1B(FBDLE)         ; DELETE ON LAST CLOSE
36        000400 .DUSR    FBFUB=  1B(FBFDB)         ; FILE HAS A USER DATA AREA
37        000200 .DUSR    FBFLF=  1B(FBFLB)         ; FLUSH TO FIB
38        000100 .DUSR    FBEXO=  1B(FBOEX)         ; FILE EXCLUSIVELY OPENED
39        000040 .DUSR    FBPIO=  1B(FBIOP)         ; FILE HAS I/O IN PROGRESS
40        000002 .DUSR    FBRMK=  1B(FBARA)         ; UNIVERSAL READ
41        000001 .DUSR    FBEMK=  1B(FBAEA)         ; UNIVERSAL EXECUTE
42
43
44                ;
45                ;  FCB STATUS BIT POINTERS
46                ;
47
48        000040 .DUSR    BFBSH=  FBSTS*SCBPW+FBSHB       ; FILE IS SHARED
49        000041 .DUSR    BFBMD=  FBSTS*SCBPW+FBMDB       ; FILE MODIFIED
50        000042 .DUSR    BFBDM=  FBSTS*SCBPW+FBDMB       ; DUMPED
51        000043 .DUSR    BFBLK=  FBSTS*SCBPW+FBLKB       ; PEND LOCK
52        000044 .DUSR    BFBTP=  FBSTS*SCBPW+FBTPB       ; TASK PENDED
53        000045 .DUSR    BFBPR=  FBSTS*SCBPW+FBPRM       ; PERMANENT FILE
54        000046 .DUSR    BFBDL=  FBSTS*SCBPW+FBDLE       ; DELETE ON LAST CLOSE
55        000047 .DUSR    BFBFD=  FBSTS*SCBPW+FBFDB       ; USER DATA AREA
          000050 .DUSR    BFBFL=  FBSTS*SCBPW+FBFLB       ; FLUSH TO FIB
          000051 .DUSR    BFBEO=  FBSTS*SCBPW+FBOEX       ; FILE EXCLUSIVELY OPENED
58        000052 .DUSR    BFPIO=  FBSTS*SCBPW+FBIOP       ; IOP POINTER
59        000056 .DUSR    BFBAR=  FBSTS*SCBPW+FBARA       ; UNIVERSAL READ
60        000057 .DUSR    BFBAE=  FBSTS*SCBPW+FBAEA       ; UNIVERSAL EXECUTE
```

```
 0018  PARFS
01
02
03                ;
04                ; MASK OF BITS ALSO IN FIB
05                ;
06
07        121037 .DUSR    FISCM=  FBDMP+FBSHR+FBDEL+APMSK
08        123437 .DUSR    FISTM=  FISCM+FBPRF+FBFUB
09
10
11
12                ;
13                ; FIB STATUS BIT POINTERS
14                ;
15
16        000142 .DUSR    BFIDM=  FISTS*SCBPW+FBDMB       ; DUMPED
17        000145 .DUSR    BFIPM=  FISTS*SCBPW+FBPRM       ; PERMANENT FILE
18        000146 .DUSR    BFIDL=  FISTS*SCBPW+FBDLE       ; DELETE FILE ON LAST CLO
19        000147 .DUSR    BFIBD=  FISTS*SCBPW+FBFUB       ; USER DATA AREA
```

```
!0019  PARFS
01                ;
02                ; LOGICAL UNIT CONTROL BLOCK PARAMETERS
03                ; THERE IS ONE LCB PER LOGICAL DISK
04                ; IN THE SYSTEM. ALL PHYSICAL UNITS WITHIN
                  ; A LDU ARE TREATED AS A SINGLE ADDRESS SPACE.
                  ;
07
08        000000 .DUSR    LBUDP=   0        ;POINTER TO UNIT DEFINITION BLOCK(UDB)
09        000001 .DUSR    LBCLP=   1        ;CACHE LIST POINTER
10        000002 .DUSR    LBTM2=   2        ;(UNUSED)
11        000003 .DUSR    LBLBP=   3        ;LCB LIST PTR(LIST OF ALL LCBS)
12        000004 .DUSR    LBTM3=   4        ;(UNUSED)
13        000005 .DUSR    LBTM4=   5        ;(UNUSED)
14        000006 .DUSR    LBBML=   6        ;LCB BIT MAP LOCK(0=>FREE)
15        000007 .DUSR    LBTM5=   7        ;(UNUSED)
16        000010 .DUSR    LBMFC=   10       ;BIT MAP FCB REAL ADDRESS
17        000011 .DUSR    LBVMF=   11       ;BIT MAP FCB VIRTUAL ADDRESS
18        000012 .DUSR    LBMBF=   12       ;BIT MAP BUFFER ADDRESS
19        000013 .DUSR    LBRCB=   13       ;LDU'S ROOT CCB ADDRESS OR 0
20        000014 .DUSR    LBCSH=   14       ;CURRENT SIZE (HIGH)
21        000015 .DUSR    LBCSL=   15       ;(LOW)
22        000016 .DUSR    LBMSH=   16       ;MAX SIZE(HIGH)
23        000017 .DUSR    LBMSL=   17       ;(LOW)
24        000020 .DUSR    LBBLT=   LBMSL-LBUDP+1    ;LCB LENGTH
```

```
:0020  PARFS
01                ;
02                ; INPUT/OUTPUT CONTROL BLOCK PARAMETERS(IOCB)
03                ; THIS BLOCK IS USED BY THE SYSTEM AS THE TEMPORARY VARIABLES
04                ; NEED TO RUN A CCB REQUEST.
05                ;
06
07        000000 .DUSR    IONQB=  0         ;NQBLK ADDRESS(DEMAND PAGING)
08        000001 .DUSR    IOCCB=  1         ;CCB ADDRESS
09        000002 .DUSR    IOFWD=  2         ;IOCB LINK
10        000003 .DUSR    IOBAK=  3         ;IOCB LINK(BACKWARDS)
11        000004 .DUSR    IOSTW=  4         ;IOCB STATUS
12        000005 .DUSR    IOSPC=  5         ;SAVED PC
13                        ;THE SAVE LEVELS ARE USED BY SUBROUTINES TO HOLD
14                        ;RETURN ADDRESSES
15        000006 .DUSR    IOSL0=  6         ;SAVE LEVEL 0
16        000007 .DUSR    IOSL1=  7         ;SAVE LEVEL 1
17        000010 .DUSR    IOSL2=  10        ;LEVEL 2
18        000011 .DUSR    IOSL3=  11        ;3
19        000012 .DUSR    IOSL4=  12        ;4
20        000013 .DUSR    IOERR=  13        ;HOLDS ERROR CODE
21        000014 .DUSR    IOLBC=  14        ;LAST BLK CORRECTION(USED IN READ EOFS)
22                        ;THE FOLLOWING IS A BUFFER HEADER THAT DSKIO
23                        ;USES TO NQ TO BUFIO
24                        ;(SEE PARS FOR BUFFER HEADER FORMAT)
25        000015 .DUSR    IOADR=  15        ;DATA ADDRESS
26        000016 .DUSR    IOST=   16        ;STATUS
27        000017 .DUSR    IOQLK=  17        ;LINK TO NEXT BH
28        000020 .DUSR    IONBL=  20        ;#BLKS
29        000021 .DUSR    IOMAP=  21        ;MAP-COMES FROM CBMAP
30        000022 .DUSR    IOUPD=  22        ;UNPEND(POST PROCESS) ADDRESS
31        000023 .DUSR    IOTCB=  23        ;TCB ADDRESS
32        000024 .DUSR    IODAH=  24        ;DATA ADDRESS(HIGH)
33        000025 .DUSR    IODAL=  25        ;(LOW)
34
35                ; MORE VARIABLES
36
37        000026 .DUSR    IOTPC=  26        ;COUNTER OF #LEVELS OF INDEXING NEEDED
38        000027 .DUSR    IOIXH=  27        ;INDEXING WORD(HIGH).HAS <0><X1>
39        000030 .DUSR    IOIXL=  30        ;(LOW). HAS <X2><X3>
40        000031 .DUSR    IONLV=  31        ;CURRENT #LEVELS IN FILE
41        000032 .DUSR    IOVAR=  32        ;TEMP VARIABLE
42        000032 .DUSR    IOICB=  IOVAR     ;HOLD IOCB ADDRESS THROUGH BUFIO
43                                          ;DO NOT MOVE IOICB, RELATIVE TO
44                                          ;IOADR!!!!
45        000033 .DUSR    IOQHI=  33        ;Q=FILE ELEMENT #=BLK#/ELEMENT SIZE
46        000034 .DUSR    IOQLO=  34        ;Q(LOW)
47        000035 .DUSR    IOREH=  35        ;REMAINDER FROM Q COMPUTATION(HIGH)
48        000036 .DUSR    IOREM=  36        ;REMAINDER FROM Q COMPUTATION
49        000037 .DUSR    IODEH=  37        ;DATA ELEMENT SIZE(HIGH) OF FILE
50        000040 .DUSR    IODES=  40        ;DATA ELEMENT SIZE(LOW)
51        000041 .DUSR    IODT1=  41        ;DOUBLE PRECISON TEMP
52        000042 .DUSR    IODT2=  42        ;(LOW)
53        000043 .DUSR    IOTP1=  43        ;ANOTHER TEMP.
54        000044 .DUSR    IORBC=  44        ;AMOUNT OF BYTES TRANSFERRED SO FAR
55        000045 .DUSR    IOFCB=  45        ;HOLDS MAPPED FCB ADDRESS
56        000046 .DUSR    IOVFC=  46        ;HOLDS VIRTUAL FCB ADDRESS
57        000047 .DUSR    IOBFA=  47        ;HOLDS BUFFER HEADER ADDRESS
58        000050 .DUSR    IOTBK=  50        ;HOLDS BLOCK COUNT
59
60        000100 .DUSR    IOLTH=  100       ;IOCB LENGTH = 100 TO AVOID GSMEM FRAGME
```

```
 0021  PARFS
01
02                ;
03                ; IOCB STATUS CODES
04                ;

        000000 .DUSR    IODOR=  0       ;IOCB DORMANT
07      177777 .DUSR    IOWAT=  -1      ;WAITING FOR I/O
08      000001 .DUSR    IORDY=  1       ;IOCB READY
09      000002 .DUSR    IOBWT=  2       ;WAITING FOR A BUFFER
10      000003 .DUSR    IOBMW=  3       ;WAITING ON THE BIT MAP
11      000004 .DUSR    IOFWT=  4       ;WAITING ON FILE IOP
12
13                ;
14                ; DELETE DATA BLOCK-USED WHEN DELETING A FILE
15                ;
16
17      000000 .DUSR    DDSPC=  0       ;SAVE PC
18      000001 .DUSR    DDBFA=  1       ;BUFFER HEADER ADDRESS
19      000002 .DUSR    DDLVL=  2       ;LEVEL COUNTER
20      000003 .DUSR    DDTPC=  3       ;TEMP COUNTER
21      000004 .DUSR    DDBLT=  DDTPC-DDSPC+1   ;BLOCK SIZE
22      000014 .DUSR    IOSTK=  IOLBC   ;START OF STACK AREA
23      000043 .DUSR    IODDP=  IOTP1   ;USED AS STACK POINTER
24
25                      ;DELETES ARE DONE IN A RECURSIVE MANNER. ONE
26                      ;OF THE ABOVE BLOCKS IS USED FOR EACH LEVEL
27                      ;OF THE FILE. SINCE DSKIO HAS NO STACK, A REDEFINED PART
28                      ;IOCB IS USED
29
30
```

```
)022  PARFS
1                ;
2                ;  UNIT DEFINITION BLOCK PARAMETERS
3                ;
4
5       000000 .DUSR   UDDCT=   0        ; DCT ADDRESS
6       000001 .DUSR   UDUNT=   1        ; DEVICE UNIT NUMBER
7       000002 .DUSR   UDCRQ=   2        ; UNIT REQUEST LIST
8       000003 .DUSR   UDLAH=   3        ; LAST LOGICAL ADDRESS (HI)
9       000004 .DUSR   UDLAL=   4        ; LAST LOGICAL ADDRESS (LO)
0       000005 .DUSR   UDUDL=   5        ; UDB FORWARD LINK (LOGICAL)
1       000006 .DUSR   UDUDP=   6        ; UDB FOREWARD LINK (PHYSICAL)
2       000007 .DUSR   UDFAH=   7        ; UNIT START ADDR (LESS FUDGE FOR INVIS A
3       000010 .DUSR   UDFAL=   10       ; UNIT START ADDRESS (LOW)
4       000011 .DUSR   UDNBK=   11       ; NUMBER OF BLOCKS TO MOVE
5       000012 .DUSR   UDSTS=   12       ; UNIT STATUS WORD
6       000013 .DUSR   UDCYS=   13       ; CYLINDER SIZE IN SECTORS (UDNSC*UDNHD)
7       000014 .DUSR   UDNSC=   14       ; NUMBER OF SECTORS PER TRACK
8       000015 .DUSR   UDNHD=   15       ; FLAGS (LEFT BYTE), NUMBER OF HEADS (RIG
9       000016 .DUSR   UDDAH=   16       ; DATA ADDRESS
0       000017 .DUSR   UDDAL=   17       ; DATA ADDRESS (LOW)
1       000020 .DUSR   UDDOA=   20       ; DOA WORD (TEMP)
2       000021 .DUSR   UDDOC=   21       ; DOC WORD-DONT REDEFINE!-....
3                                        ; USED AS RUNNING CYLINDER NUMBER
4       000022 .DUSR   UDERC=   22       ; ERROR COUNTER
5       000023 .DUSR   UDTBK=   23       ; TEMP BLOCK COUNTER
6       000024 .DUSR   UDERF=   24       ; ERROR FLAGS
7       000025 .DUSR   UDUNS=   25       ; UNIT STATUS (DIA OR DIC ONLY)
8       000026 .DUSR   UDBBT=   26       ; BAD BLOCK TABLE POINTER
9       000027 .DUSR   UDRMH=   27       ; REMAPPED ADDRESS HI
0       000030 .DUSR   UDRML=   30       ; REMAPPED ADDRESS LO
1
2       000033 .DUSR   UDSIB=   33       ;UNIT STATUS (DIB FOR ZEBRA)
3
4                      ;THE FOLLOWING USED FOR FATAL (HARD) ERRORS
5                      ;       NEED TO BE SAVED INCASE UDB RESTARTED ON DIFFEREN
6                      ;       TRANSFER PRIOR TO PANICING, ETC
7
8       000034 .DUSR   UDEOA=   34       ;DOC (READ/WRITE) ON FATAL ERROR
9       000035 .DUSR   UDEOC=   35       ;CYLINDER # ON FATAL ERROR
0       000036 .DUSR   UDDIA=   36       ;DIA OR DIC ON FATAL ERROR
1       000037 .DUSR   UDDIB=   37       ;DIB ON FATAL ERROR
2
3                ;MAG TAPE UNIT DIFFERENCES
4
5       000033 .DUSR   UDST1=   UDSIB    ; STATUS REGISTER 2(DIC). USED ON 6026, D
6       000013 .DUSR   UDWFG=   UDCYS    ; WRITE FLAG FOR CLOSE
7       000023 .DUSR   UDERS=   UDTBK    ; ERASE COUNT (AFTER BAD TAPE)
8                ;THE NEXT 6 WORDS MUST BE CONTIGIOUS AND ORDERED WITH THE FCB
9       000014 .DUSR   UDLFL=   UDNSC    ; LOGICAL EOT FILE
0       000015 .DUSR   UDLBL=   UDNHD    ; LOGICAL EOT BLOCK
1       000016 .DUSR   UDFIL=   UDDAH    ; CURRENT FILE
2       000017 .DUSR   UDBLK=   UDDAL    ; CURRENT BLOCK
3       000020 .DUSR   UDOBL=   UDDOA    ; MOST RECENT OLD BLOCK COUNT
4       000021 .DUSR   UDOB2=   UDDOC    ; SECOND MOST RECENT OLD BLK CNT
5
6                ;      THE FOLLOWING ARE USED BY UNERR
7
8       000031 .DUSR   UDEST=   31       ; ERROR STATUS
9       000032 .DUSR   UDRTY=   32       ; RETRYS AFTER ERROR
0                                        ; 180 => ERROR
```

```
 0023  PARFS
01                                      ; 1B1 => HARD
02                                      ; 1B2 => DATA LATE (SPECIAL TYPEOUT NEEDE
03
04        000040 .DUSR    UDBLT=  UDDIB-UDDCT+1   ;UDB ENTRY LENGTH
05
06              ; MOVING HEAD DISK UDB STATES (UDSTS)
07
08
09        000000 .DUSR    DPIDL=  0       ;IDLE
10        000001 .DUSR    DPSKR=  DPIDL+1 ;SEEK READY
11        000002 .DUSR    DPSKP=  DPSKR+1 ;SEEK IN PROG
12        000003 .DUSR    DPSKD=  DPSKP+1 ;SEEK DONE
13        000004 .DUSR    DPIOR=  DPSKD+1 ;I/O READY
14        000005 .DUSR    DPIOP=  DPIOR+1 ;I/O IN PROG
15        000006 .DUSR    DPIOD=  DPIOP+1 ;I/O DONE
16        000007 .DUSR    DPRCR=  DPIOD+1 ;RECAL READY
17        000010 .DUSR    DPRCP=  DPRCR+1 ;RECAL IN PROG
18        000011 .DUSR    DPRCD=  DPRCP+1 ;RECAL DONE
19        000012 .DUSR    DPSKE=  DPRCD+1 ;SEEK ERROR
20        000013 .DUSR    DPIOE=  DPSKE+1 ;I/O ERROR
21        000014 .DUSR    DPFTE=  DPIOE+1 ;FATAL ERROR
22        000015 .DUSR    DPGST=  DPFTE+1 ;GET STATUS
23
24              ; DUAL MODE TAPE BIT POINTERS.
25
26        000676 .DUSR    UDSFE=  UDST1*SCBPW+14. ;BIT POINTER TO FORMAT ERROR.
27        000677 .DUSR    UDSPE=  UDST1*SCBPW+15. ;BIT POINTER TO FORMAT ERROR.
28        000250 .DUSR    UDSBT=  UDSTS*SCBPW+8.  ;BIT POINTER TO BOT.
29
```

```
10024  PARFS
01                ;
02                ; COMMON DEVICE CONTROL TABLE (DCT) PARAMETERS
03                ;
04                ;
05                ; ***************************************************
06                ; *                                                 *
07                ; *      A CHANGE TO THE COMMON AND/OR CHARACTER DCT     *
08                ; *      OFFSETS IMPLIES IOPMGR RE-ASSEMBLY         *
09                ; *                                                 *
10                ; ***************************************************
11                ;
12
13        000000 .DUSR    DCMPS=  0         ; ADDRESS OF MAP SAVE ROUTINE
14        000001 .DUSR    DCMSK=  DCMPS+1   ; INTERRUPT PRIORITY MASK
15        000002 .DUSR    DCCRQ=  DCMSK+1   ; REQUEST QUEUE POINTER
16        000003 .DUSR    DCINS=  DCCRQ+1   ; ADDRESS OF INTERRUPT SERVICE ROUTINE
17        000004 .DUSR    DCSTR=  DCINS+1   ; DEVICE START ROUTINE ADDRESS
18                                          ;  (INPUT START FOR CHARACTER DEVICES)
19        000005 .DUSR    DCDVC=  DCSTR+1   ; DEVICE CODE
20        000006 .DUSR    DCSTS=  DCDVC+1   ; STATUS
21        000007 .DUSR    DCPRS=  DCSTS+1   ; POWERFAIL RESTART ROUTINE
22
23                ; ERCC REDEFINTIONS
24
25        000005 .DUSR    DCERC=  DCDVC     ; ERCC FAULT CODE
26        000006 .DUSR    DCERF=  DCSTS     ; ERCC FAULT ADDRESS
27
28                ; POWERFAIL REDEFINITIONS
29
30        000002 .DUSR    DCDAT=  DCCRQ     ; DATE OF POWERFAIL
31        000004 .DUSR    DCTMH=  DCSTR     ; TIME (H) OF POWERFAIL
32        000006 .DUSR    DCTML=  DCSTS     ; TIME (L) OF POWERFAIL
33                ;
34                ; BLOCK I/O DEVICE CONTROL TABLE PARAMETERS
35                ;
36
37        000010 .DUSR    DCNMS=  DCPRS+1   ; NUMBER OF MAP SLOTS REQUIRED
38        000011 .DUSR    DCFMS=  DCNMS+1   ; HIGHEST MAP SLOT DEVICE CAN USE (32. OR
39                                          ; INIT CHANGES TO ASSIGNED MAP SLOT
40        000012 .DUSR    DCPSL=  DCFMS+1   ; FIRST MAP SLOT FOR PAGING DISK G-TABLE
41        000012 .DUSR    DCCMD=  DCPSL     ; LAST COMMAND
42        000012 .DUSR    DCFLG=  DCPSL     ; RECV STATE (MCA RECVR)
43        000013 .DUSR    DCPUL=  DCPSL+1   ; PHYSICAL UNIT DEFINITION (UDB) LIST
44        000014 .DUSR    DCSUP=  DCPUL+1   ; SETUP ROUTINE ADDRESS
45        000014 .DUSR    DCLMA=  DCSUP     ; LAST MAPPED ADDRESS (MAG TAPE)
46        000015 .DUSR    DCTIU=  DCSUP+1   ; INIT UDB ROUTINE
47        000015 .DUSR    DCDBH=  DCTIU     ; DUMMY BUFFER HEADER
48        000016 .DUSR    DCENQ=  DCTIU+1   ; ENQUEUE ROUTINE
49        000016 .DUSR    DCQVT=  DCENQ     ; QUEUE VECTOR TABLE
50        000017 .DUSR    DCLNK=  DCENQ+1   ; UNIT DCT CHAIN LINK
51        000020 .DUSR    DCCTO=  DCLNK+1   ; CHECK TIMEOUT BLOCK ROUTINE
52        000020 .DUSR    DCQTB=  DCCTO     ; QUEUE TABLE
53        000027 .DUSR    DCMQB=  27        ; MAPPED QUEUE TABLE ADR
54        000021 .DUSR    DCTYP=  DCCTO+1   ; TYP OF PHYSICAL UNIT (I.E. 6060,6063,TA
55        000022 .DUSR    DCODC=  DCTYP+1   ; ADDR OTHER DCT (MCA)
56
57                ; LPB REDEFINITIONS
58
59        000012 .DUSR    DCTAB=  DCCMD     ; TAB FORMAT ROUTINE
60        000014 .DUSR    DCRST=  DCSUP     ; TAB RECOVERY ROUTINE
```

```
01
02              ; TIME OUT BLOCK EXTENSION
03
04      000022 .DUSR    DCLKF=  DCTYP+1 ; LINK FORWARD
        000023 .DUSR    DCLKB=  DCLKF+1 ; LINK BACKWARD
        000024 .DUSR    DCCNT=  DCLKB+1 ; COUNT (SECONDS)
07      000025 .DUSR    DCMSG=  DCCNT+1 ; MESSAGE TO ROUTINE (DCT)
08      000026 .DUSR    DCRTN=  DCMSG+1 ; TIME OUT ROUTINE
```

```
01
02                ; DATA LATE EXTENSION
03
04      000030 .DUSR    DCDLC=  30      ;# OF DATA LATES RETRIES BEFORE REPORTING
05      000031 .DUSR    DCDLD=  DCDLC+1 ;DATE LAST REPORT
06      000032 .DUSR    DCDLT=  DCDLD+1 ;TIME LAST REPORT (BISECONDS)
07
08                ; LENGTHS OF VARIOUS DISK DCT'S
09
10      000033 .DUSR    DCPDL=  DCDLT+1 ;LENGTH OF FLOPPY/TOP-LOADER DCT
11      000033 .DUSR    DCPEL=  DCPDL   ;LENGTH OF 3330 DCT
12
13                ; EXTENSION USED BY ECC FEATURE OF THE ZEBRA AND PAGING DSK DRIV
14
15      000033 .DUSR    DCWD0=  DCDLT+1 ;FIRST ERROR WORD (IC)
16      000034 .DUSR    DCWD1=  DCWD0+1 ;SECND ERROR WORD (IP)
17      000035 .DUSR    DCWDD=  DCWD1+1 ;WORD DISPLACEMENT
18      000036 .DUSR    DCECT=  DCWDD+1 ;ITERATION TEMP
19      000037 .DUSR    DCEC1=  DCECT+1 ;FIRST ECC REGISTER
20      000040 .DUSR    DCEC2=  DCEC1+1 ;SECND ECC REGISTER
21      000041 .DUSR    DCEC3=  DCEC2+1 ;CORE ADDR FROM STRTIO RTN
22      000042 .DUSR    DCEC4=  DCEC3+1 ;MEMORY ADDR AT INTRPT TIME
23      000043 .DUSR    DCP1=   DCEC4+1 ;TEMP
24
25      000041 .DUSR    DCQAR=  DCEC3   ;CONTENTS OF QUEUE ADDR REG(PGING DSK)
26      000042 .DUSR    DCSEC=  DCEC4   ;NUMBER OF SECTORS MOVED(PGING DSK)
27
28                ; MORE LENGTHS
29
30      000044 .DUSR    DCPFL=  DCP1+1  ;LENGTH OF ZEBRA DCT
31      000044 .DUSR    DCKBL=  DCPFL   ;LENGTH OF PAGING DISK DCT
32
33                ;
34                ; CHARACTER ORIENTED DEVICE DCT PARAMETERS
35                ;
36
37      000010 .DUSR    DCPTA=  DCPRS+1 ; PT ADDRESS OF PMGR FOR DEVICE
38      000011 .DUSR    DCPIB=  DCPTA+1 ; POINTER TO PIB IN PMGR (1B0 = 0)
39      000011 .DUSR    DCLTP=  DCPIB   ; PTR TO LINE TABLE IN PMGR (1B0 = 1)
40      000012 .DUSR    DCPMP=  DCPIB+1 ; DOA MAP WORD FOR PIB (OR LTBL) ADDRESS
41                                      ;       = 0 => NO PMGR
42      000013 .DUSR    DCOST=  DCPMP+1 ; OUTPUT START ROUTINE
43
44      000014 .DUSR    DCCHL=  DCOST+1 ; LENGTH OF CHARACTER DEVICE DCT
45
46                ; EXTENSIONS USED BY DCU AND IOP
47
48      000014 .DUSR    DCPSV=  DCOST+1 ; POWERFAIL SAVE ROUTINE
49      000015 .DUSR    DCPST=  DCPSV+1 ; POWERFAIL START ROUTINE
50      000016 .DUSR    DCHPC=  DCPST+1 ; HALTED PC
51
52                ; DCU EXTENSION (USED BY POWERFAIL)
53
54      000017 .DUSR    DCMPL=  DCHPC+1 ; DCH MAP WORDS USED BY DCUSV
55      000023 .DUSR    DCMPI=  DCMPL+4 ; DCH MAP WORD USED BY PWRRS
56      000024 .DUSR    DCMPR=  DCMPI+1 ; DCH MAP WORDS USED BY DCULD
57      000031 .DUSR    DCLFG=  DCMPR+5 ; DCU LOAD FLAG
58      000032 .DUSR    DCSVA=  DCLFG+1 ; DCU SAVE AREA, MUST CORRESPOND
59                                      ;  TO SIZE OF DCU VOLATILE AREA
60
```

```
0027  PARFS
01
02              ; LENGTH OF DCU DCT
03
04      000046 .DUSR    DCDCL=  DCSVA+10
05
06              ; PWRDC EXTENSION
07
08      000010 .DUSR    DCPSP=  DCPRS+1 ; POWERFAIL SP
09      000011 .DUSR    DCPFP=  DCPSP+1 ; POWERFAIL FP
10      000012 .DUSR    DCM70=  DCPFP+1 ; CUR70 (SAVED BY PWRFL)
11      000013 .DUSR    DCM72=  DCM70+1 ; CUR72 (SAVED BY PWRFL)
12      000014 .DUSR    DCM76=  DCM72+1 ; CUR76 (SAVED BY PWRFL)
13      000015 .DUSR    DCPTM=  DCM76+1 ; TEMP
14      000016 .DUSR    DCPCN=  DCPTM+1 ; COUNT OF SPURIOUS INTERRURTS
15      000017 .DUSR    DCDCA=  DCPCN+1 ; ADDRESS DCUDC
16      000020 .DUSR    DCIPA=  DCDCA+1 ; ADDRESS IOPDC
17      000021 .DUSR    DCDMA=  DCIPA+1 ; ADDRESS OF DCH A MAP TABLE
18      000022 .DUSR    DCDMB=  DCDMA+1 ; ADDRESS OF DCH B MAP TABLE
19      000023 .DUSR    DCDMC=  DCDMB+1 ; ADDRESS OF DCH C MAP TABLE
20      000024 .DUSR    DCDMD=  DCDMC+1 ; ADDRESS OF DCH D MAP TABLE
21
22              ; IOP EXTENSION
23
24      000017 .DUSR    DCHA0=  DCHPC+1 ; HALTED AC0
```

```
3028  PARFS
1                ;
2                ; DCT STATUS BIT POSITIONS
3                ;
4
5       000000 .DUSR    DCPER=  0        ; PERIPHERAL DEVICE
6                ;
7       000002 .DUSR    DCLOK=  2        ; DCT LOCK
8       000003 .DUSR    DCPRO=  3        ; PROTOCOL MAPPING (MCA)
9       000004 .DUSR    DCDCU=  4        ; DCU
0       000005 .DUSR    DCTRQ=  5        ; LPB TAB REQUEST
1
.2      000006 .DUSR    DCVFU=  6        ; LPB VFU LOAD REQUEST (LPB)
.3      000006 .DUSR    DCNOV=  6        ; NO OVERLAPPED OPERATIONS (DISKS)
14
15      000007 .DUSR    DCHSC=  7        ; HIGH SPEED CHANNEL INDICATOR
16      000010 .DUSR    DCPAG=  10       ; PAGING DISK QTBL MAPPING....
17                                       ; USED BY SINIT AND MLDUI
18      000011 .DUSR    DCMSA=  11       ; MAP SLOTS ALLOCATED
19      000012 .DUSR    DCDFF=  12       ;INHIBIT INITIAL FORM FEED
20      000013 .DUSR    DCIOP=  13       ; IOP
21      000014 .DUSR    DCPWR=  14       ; POWERFAIL FLAG
22
23
24               ;
25               ; DCT STATUS BIT MASKS
26               ;
27
28      100000 .DUSR    DMPER=  1B(DCPER)        ; PERIFPHERAL DEVICE
29               ;
30      020000 .DUSR    DMLOK=  1B(DCLOK)        ; DCT LOCK
31      010000 .DUSR    DMPRO=  1B(DCPRO)        ; MCA PROTOCOL MASK
32      004000 .DUSR    DMDCU=  1B(DCDCU)        ; DCU
33      002000 .DUSR    DMTRQ=  1B(DCTRQ)        ; LPB TAB REQUEST
34      001000 .DUSR    DMVFU=  1B(DCVFU)        ; LPB VFU LOAD REQUEST (LPB)
35      001000 .DUSR    DMNOV=  1B(DCNOV)        ; NO OVERLAPPED OPERATIONS (DISKS
36      000400 .DUSR    DMHSC=  1B(DCHSC)        ; HIGH SPEED DATA CHANNEL IND.
37      000200 .DUSR    DMPAG=  1B(DCPAG)        ;PAGING DISK QTBL MAPPING
38      000100 .DUSR    DMMSA=  1B(DCMSA)        ;MAP SLOTS ALLOCATED
39      000040 .DUSR    DMDFF=  1B(DCDFF)        ;INHIBIT INITIAL FORM FEED
40      000020 .DUSR    DMIOP=  1B(DCIOP)        ; IOP
41      000010 .DUSR    DMPWR=  1B(DCPWR)        ; POWERFAIL FLAG
42
43
44               ;
45               ; DCT STATUS BIT POINTERS
46               ;
47
48      000140 .DUSR    BDCPR=  DCSTS*SCBPW+DCPER        ; PERIPHERAL DEVICE
49               ;
50      000142 .DUSR    BDLOK=  DCSTS*SCBPW+DCLOK        ; DCT LOCK
51      000143 .DUSR    BDPRO=  DCSTS*SCBPW+DCPRO        ; PROTOCOL I/O
52      000144 .DUSR    BDDCU=  DCSTS*SCBPW+DCDCU        ; DCU
53      000145 .DUSR    BDTRQ=  DCSTS*SCBPW+DCTRQ        ; LPB TAB REQUEST
54      000146 .DUSR    BDVFU=  DCSTS*SCBPW+DCVFU        ; LPB VFU LOAD REQUEST
55      000146 .DUSR    BDNOV=  DCSTS*SCBPW+DCNOV        ; NO OVERLAPPED OPS (DIS
56      000147 .DUSR    BDHSC=  DCSTS*SCBPW+DCHSC        ; HIGH SPEED CHANNEL
57      000150 .DUSR    BDPAG=  DCSTS*SCBPW+DCPAG        ;PAGING DISK QTBL MAP
58      000151 .DUSR    BDMSA=  DCSTS*SCBPW+DCMSA        ;MAP SLOTS ALLOCATED
59      000152 .DUSR    BDDFF=  DCSTS*SCBPW+DCDFF        ;INHIBIT INITIAL FORM
60      000153 .DUSR    BDIOP=  DCSTS*SCBPW+DCIOP        ; IOP
```

```
0029  PARFS
01        000154 .DUSF   BUPWR=  DCSTS*SCBPW+DCPWR          ; POWERFAIL FLAG
```

```
!0030  PARFS
01                ; DCT DISK, TAPE, AND MCA TYPE DEFINTIONS
02                ;
03                ; THESE ARE DIVIDED INTO TWO CATEGORIES:
04                ;       1) STANDARD DATA CHANNEL DEVICES- DEVICES WHICH MUST GO
05                ;          ON THE STANDARD DATA CHANNEL
06                ;       2) HIGH SPEED DATA CHANNEL(BMC) DEVICES- DEVICES WHICH
07                ;          BE ON THE HIGH SPEED CHANNEL. SINIT WILL DETERMINE IF
08                ;          IS OR NOT AT RUN TIME.
09                ;
10
11                ; HIGH SPEED (POTENTIALLY) DATA CHANNEL DEVICES
12
13        000000 .DUSR    D6060=  0       ;6060 SERIES (ZEBRA) 50MB,100MB 200MB
14        000001 .DUSR    D6063=  1       ;6063 SERIES (PAGING) 1MB,2MB FIXED HEAD
15
16                ; HIGH SPEED CHANNEL RANGES:
17
18        000000 .DUSR    HSCTL=  0       ;HIGH SPEED CHANNEL DEVICE TYPE MINIMUM
19        000001 .DUSR    HSCTH=  1       ;CURRENT HIGH SPEED CHANNEL MAXIMUM
20
21
22                ; STANDARD DATA DATA CHANNEL DEVICES
23
24        000200 .DUSR    D4234=  200     ;4234/6030 SERIES(TOPLOADER/FLOPPY)
25        000201 .DUSR    D6070=  201     ;6070 SERIES (GEMINI/DBL FLOPPY)
26        000202 .DUSR    D4231=  202     ;4231A SERIES (CDC 3330)
27        000203 .DUSR    D6020=  203     ;ALL MAGTAPE CONTOLLERS UP TO THE DUAL MO
28        000204 .DUSR    D4206=  204     ;4206/4038A MCA
29        000205 .DUSR    D6097=  205     ;6097,8,9 SERIES (ECHO/QUAD FLOPPY)
30        000206 .DUSR    D6026=  206     ;DUAL MODE 800/1600 BPI.
31
32                ; STANDARD DATA CHANNEL RANGES:
33
34        000200 .DUSR    LSCTL=  200     ;STANDARD DATA CHANNEL DEVICE TYPE MIN
35        000206 .DUSR    LSCTH=  206     ;STANDARD DATA CHANNEL DEVICE TYPE MAXIMU
36
```

```
!0031  PARFS
01                ;
02                ; BUFFER CACHE DESCRIPTOR PARAMETERS
03                ;
04
          000000 CALFL= 0          ;LRU FORWARD LINK
          000001 CALBL= 1          ;LRU BACK LINK
07        000002 CABUF= 2          ;CACHE BUFFER POINTER
08                                 ;0B0 => DESC. LINKED TO FCB, 1B0 => LINKED TO LCB
09        000003 CADFL= 3          ;FCB/LCB LIST FORWARD LINK
10        000004 CADBL= 4          ;FCB/LCB LIST BACK LINK
11        000005 CADBN= 5          ;DATA BLOCK NUMBER
12        000006 CALAH= 6          ;LOGICAL ADDR HI
13        000007 CALAL= 7          ;LOGICAL ADDR LOW
14
15        000010 CALN= 8.          ;DESCRIPTOR LENGTH
16
17
18        000040 CADBI=  (CABUF*SCBPW)+0          ;BIT OFFSET OF FCB/LCB INDICATOR
```