B U R R O U G H S

L A R G E S Y S T E M S

M C P M A N U A L

R E L E A S E 3.1/3.2

REVISION DATE: SEPTEMBER, 1981

PREFACE

This manual was written to be used in conjunction with the MCP course, and considerable time has been devoted to its preparation. Care has been taken to explain, to some degree, all subjects covered in the MCP course but some special topics have been omitted. The purpose of the MCP course is to provide the student with the knowledge and confidence to explore the MCP to a depth appropriate to his or her needs.

The NEWP portion of the MCP course (and this manual) covers the NEWP language constructs that are not included in ALGOL. The MCP portion provides structural overviews of system initialization, process control, memory management and I/O operations. Problem solving exercises are used to provide the student with experience in programming, debugging and executing NEWP programs. This manual was written with the assumption that the reader has met the prerequisites for the MCP course. These prerequisites are:

1. Attended ALGOL course.

2. Attended UTILITIES & FACILITIES course.

3. Attended BASIC SYSTEM SUPPORT course.

4. Currently, quite familiar with ALGOL and the Large Systems hardware.

SECTION 1

HARDWARE AND SOFTWARE OVERVIEW

GENERAL

The Burroughs B7000 information processing system is a large scale, multiprogramming and multiprocessing computing system. The hardware is controlled by the Burroughs Master Control Program, MCP, and can be tailored to the processing needs of a user by arranging central processor modules, input/output modules, and memory modules on an electronic grid, or exchange (figure 1-1), in a variety of ways depending upon the exact needs of the user. The MCP, itself, can be changed dramatically by merely setting or resetting run time options, etc. with simple operator input commands. The B7000 may be balanced by means of other operator instructions (and dynamically by the MCP) that control the interaction of the independently operating computing, input/output, and memory modules through the exchange; the throughput of the system as a whole is maximized, but the performance of no single element of the system is maximized to the neglect or detriment of others.

The key to the efficient balanced use of the system is the Burroughs Master Control Program, a unique executive software operating system that automatically makes optimum use of all system resources. It is this operating system that makes multiprogramming and multiprocessing both functional and practical by dynamically controlling system resources and by scheduling jobs in the multiprogramming mix. In use, the master control program allocates system resources to meet the needs of the programs introduced into the computer. It continually and automatically reassigns resources, starts jobs, and monitors their performance.

Further implications of the modularity and flexibility of the system are its expandability (a capacity to add hardware modules without reprogramming) and its increased reliability (and, thus, increased availability to the user). This reliability is achieved by the use of failsoft techniques that (in addition to providing for error detection and error correction, redundancy of power supplies) exclude faulty modules from the system and permit processing to continue (again, without reprogramming) even with a temporarily reduced configuration.

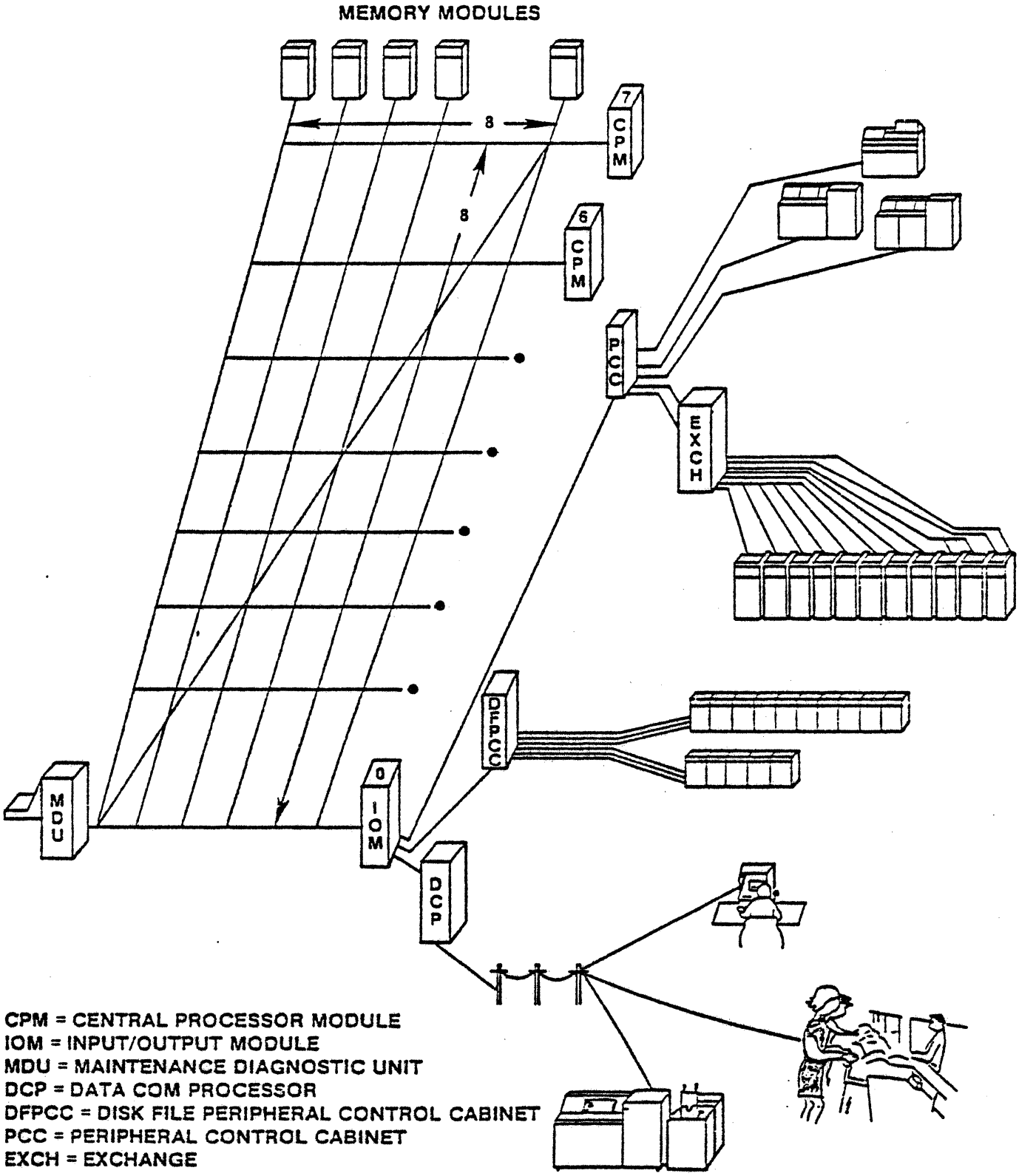


Figure 1-1. B7700 Exchange

Figure 1-1. B7000 Exchange

SYSTEM CONFIGURATION

Physically, the components of the B7000 system fall into three categories, as follows:

1. Central components of the B7000 system: the central processor module, input/output module, the memory module, the maintenance diagnostic unit, and the operator's console.

2. Standard Burroughs cabinets that contain peripheral controls and exchanges, the disk optimizer, the data communications processor, and AC power supplies.

3. Standard peripheral devices that are joined to the central system by means of standard Burroughs peripheral controls, adapters, and exchanges, and standard remote devices that are joined to the central system by means of line adapters and the data communications processor.

The arrangement of these components into a system and the size of the system depend on the application and workload of the user.

HARDWARE REVIEW

Before looking at the MCP it is important for the student to have a good understanding of the B7000 hardware. Each module on the system will be discussed. The discussion will start with a general overview of the system and follow with detailed information on each module.

Figure 1-2 is a diagram of a B7000 system. The system is composed of Central Processor Modules (CPM), Input/Output Modules (IOM), Memory Control Modules (MCM) and Data Communication Processors (DCP).

B 7000 SYSTEM CONNECTIVITY BLOCK DIAGRAM

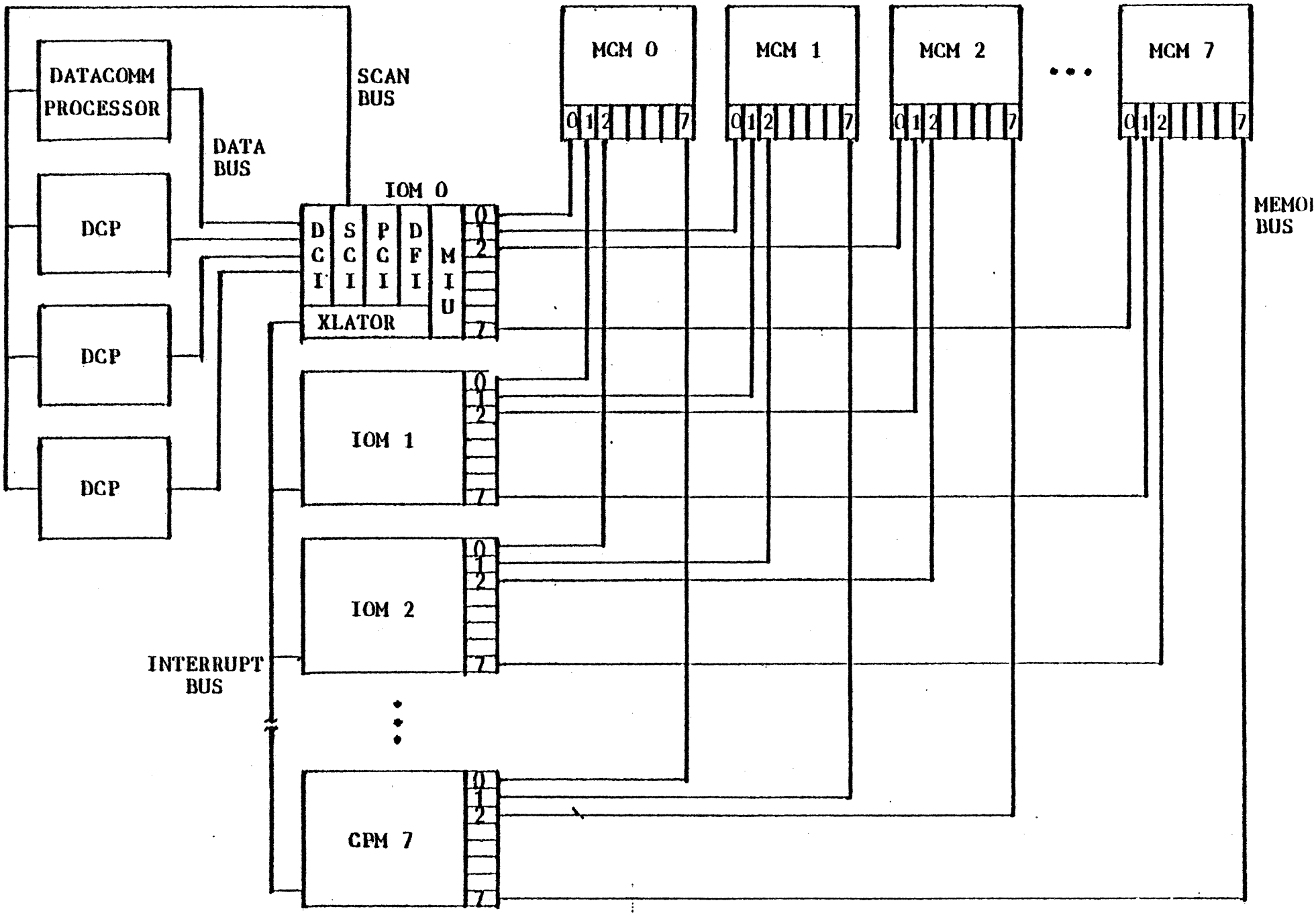


FIGURE 1-2

FIGURE 1-2. B7000 SYSTEM

The following general comments can be made about the B7000 system.

1. Each module on the system is independently powered.

2. All CPM's and IOM's are connected with an interrupt bus.

3. There is a scan bus that connects an IOM and the DCP's on that IOM.

4. The DCP's on an IOM will use the IOM's memory access logic.

5. Each CPM has a time of day clock.

6. There is one master clock for the system. It has it's own power supply.

A general discussion of each system component follows. This discussion is intended to give the student an overview of the system.

MEMORY

All memory will do single bit error correction and multi-bit error detection. Access to memory is phased (interleaved). All requestors will take advantage of memory phasing. The IOM will make four word requests to an MCM. CPM's will make four or eight word requests to an MCM. A planar memory MCM will return at most four words. A IC memory MCM will return at most eight words. Thus, the number of words returned for a given request is based on the type of memory.

CPM

The CPM has 1K word (B7700) or 2K word (B7800) local memory storage and a 32 word (B7700) or 2K word (B7800) code buffer. There is a 32 word stack buffer in the B7700 CPM. The B7800 has a 32 word store queue which will buffer data before it is sent to memory. Thus, memory accesses can be grouped into multi-word stores. Also repeated stores to an address can be eliminated, only the newest data will be written to memory. The local buffers in the B7000 CPM will attempt to keep memory requests to a minimum and will try to have code and data ready for the execution unit. Thus, the CPM can overlap fetching of data with execution of instructions.

IOM

An IOM has up to 28 fixed channels. Each channel has its own (2 four word) buffers. Data is transfered to and from the IOM in 4 word groups. The IOM does it's own path checking and can handle a queue of I/O's, not just one at a time. I/O results are reported in a queue. Thus, an IOM need not interrupt a CPM for an I/O finish. The IOM will handle conditional seek logic for disk packs. The IOM uses MCP generated I/O queues and an I/O map.

MDU (B7700)

The MDU is connected to all modules on the system. The MDU can read a test tape and use this information to check a module. The MDU is also used to take panel dumps and test cards.

MDP (B7800)

Maintenance Diagonistic Processing (MDP) is used for testing mainframe modules (CPM,IOM and MCM), testing logic cards and PROM programming. MDP can be run online with a B7800 CPM or with the Maintenance Processor (MP). The MP is a B800 system. Thus, maintenance operations can be run by a B7800 CPM or the B800. The MP can be connected to a datacomm line so remote diagnostics can be done.

MAXIMUM CONFIGURATION

```
  8 Processors (IOM and CPM)
  8 MCM's (2.5 million words usable)
255 Units
  8 DCP's
```

The following part of this review will give a more detailed discussion of the MCM, CPM and IOM. In addition, the MCP interface with the IOM will be discussed.

MEMORY

Figure 1-3 is a diagram of a planar memory MCM. A system can have up to 8 MCM's. Each MCM can have up to 2 Memory Storage Cabinets (MSC). Each MSC has 2 Memory Storage Units (MSU). A MSU has 65K words. Planar memory will phase at most 4 words. Thus, the maximum number of words that can be transfered is 4 words. Following is the amount of time required to transfer 4 words using planar memory.

1 MSC/MCM 3.75 microseconds (2 word phased)

2 MSC/MCM 2.125 microseconds (4 word phased)

Figure 1-4 is a diagram of a IC memory MCM. A system can have up to 8 MCM's. Each MCM can have up to 2 Memory Storage Units (MSU). Each MSU has 131K words. A Memory Storage Cabinet contains up to 4 Memory Storage Units. Memory is phased at the MSU level. IC memory will phase at most 8 words. Thus, the maximum number of words that can be transfered is 8 words. Following is the amount of time required to transfer the number of words indicated using IC memory.

1.87 microseconds for 4 words
2.37 microseconds for 8 words

Maximum addressable memory by a requestor is 1 million words. This is four full MCM's. A system using global memory (tightly coupled) can use 2 million words (2.5 million words with MOD III MCM).

A model III MCM can control 500K words. Thus, a full memory configuration (1 million words) could be 2 MCM's and 2 IC memory MSC's.

Memory words are 60 bits.

8 bits for error correction and detection. This allows single bit error correction and multi bit error detection.

52 data bits are transfered to and from requestor.
48 data
3 tag
1 parity

The MCM is responsible for:

Buffering multi word transfers to and from a requestor.

Locking out MCM's to requestors. This is used when system is split or running tightly coupled. Requestors are locked out by setting requestor inhibit switches on the MCM. These switches can be set by the operator or the MCP.

Controlling requestor priority.

Doing error correction and detection.

Broadcasting which addresses the MCM controls. The address range switches (UPPER and LOWER)

specify the memory addresses a MCM spans. These switches can be set by the operator or MCP. The register contains the six most significant bits of the address that is controlled by this MCM.

Maintaining the MSU status register which specifies which MSU's are available. These switches can be set by the operator or MCP.

As an example of how these switches are set:

Assume a system is configured as follows:

2 MCM 4 MSU each (planar MSU)
1 IOM (IOM 0)
1 CPM (CPM 5)

The switches should be set as follows:

MCM	LOWER	UPPER	INHIBIT	MSU STATUS
1	000000	001111	11011110	1111
2	010000	011111	11011110	1111

A requestor must keep track of how many words are received from a memory request. If all words that were requested were not received a request for the remaining words must be made. For example, if a request is made for 8 words from an MCM that is only 4 way phased, only 4 words are returned. The requestor must change the word count and memory address in the MCM control word adn make the request to the MCM again.

MCM BLOCK DIAGRAM

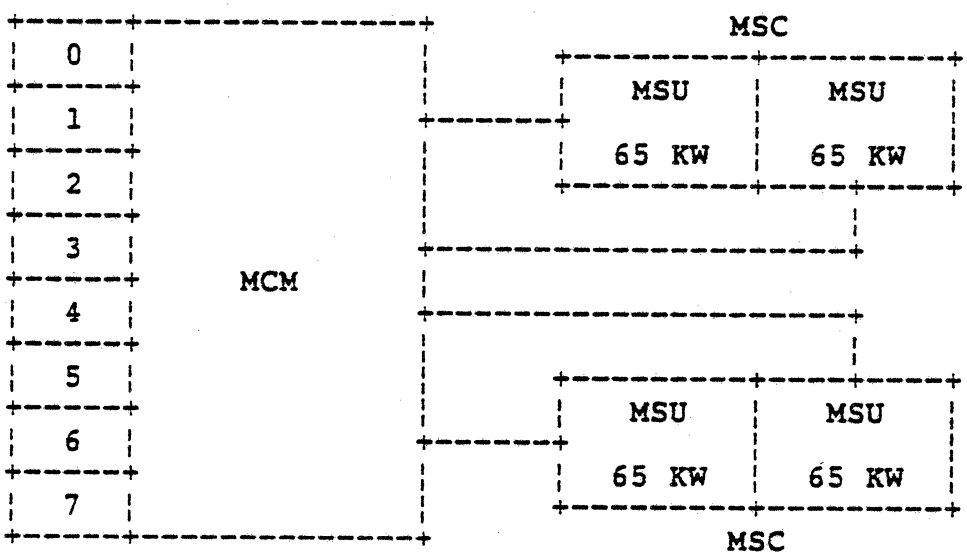


Figure 1-3. MCM Block Diagram (Planar)

MCM BLOCK DIAGRAM

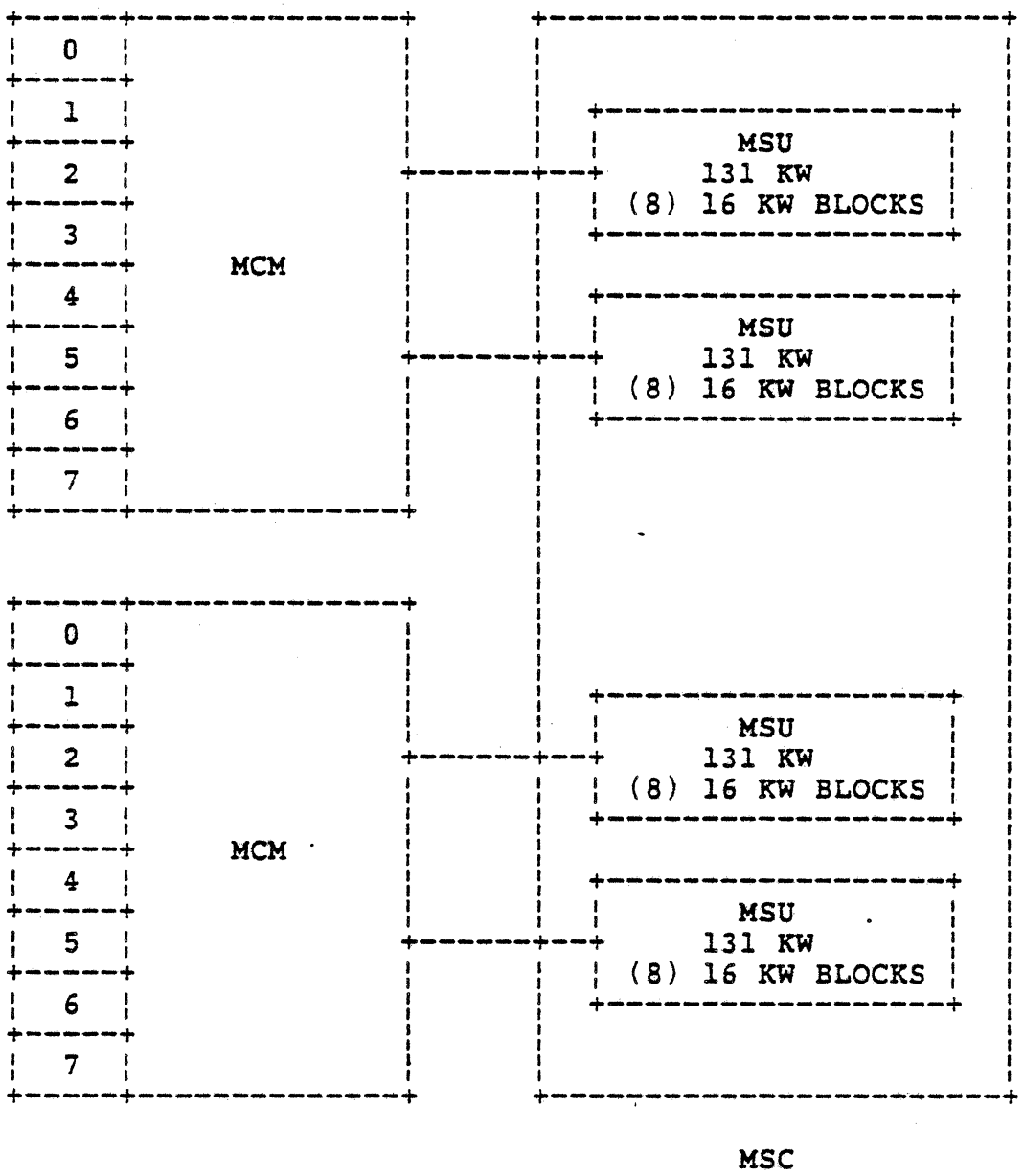


Figure 1-4. MCM Block Diagram (IC)

CPM

The B7000 CPM is composed of several independent units. It is this independence that allows data fetches to be overlaped with instruction execution. The units of the B7000 CPM are:

COMMUNICATIONS UNIT (B7700) MEMORY ACCESS UNIT (B7800):

The communications unit (B7700) and the memory access unit (B7800) provide memory interface for the CPM. These units serve the same function. However, the memory access unit is capable of doing simultaneous fetches and stores to memory as long as the addresses are not in the same MCM.

PROGRAM CONTROL UNIT:

The Program Control Unit (PCU) interprets object code and builds micro operators for the execution unit. These micro operators are placed in a queue for the execution unit. The PCU will pre process code doing operations such as taking unconditional branches, generating absolute addresses from relative addresses and operator concatenation. That is, join more than one instruction into a single micro op.

In addition, on the B7800 CPM the PCU is responsible for allocation of registers in the Central Data Buffers (CDB).

EXECUTION UNIT:

The Execution Unit (EU) performs the actual operations. These operators come from a queue built by the PCU.

On the B7800 additional logic has been added to the EU for short arithmetics, add operations less than 20 bits and multiply operations with results less than 16 bits.

STORAGE UNIT (B7700):

The storage unit will start fetch and store operations, it is driven by the PCU and execution unit.

STORE QUEUE (B7800):

All stores to main memory go thru the store queue. The store queue will try to group adjacent words into a multi word write. In addition, it will test for repeated stores to the same address. This will help cut down on

memory traffic.

LOCAL MEMORY:

Program buffer will hold up to 32 words (B7700) or 2K words (B7800) of object code in the CPM. On the B7800 the program buffer is loaded much like associative memory.

32 words of stack buffer (B7700) for top of stack information.

1K (B7700) or 2K (B7800) words of data and control information (associative memory).

DATA REFERENCE UNIT (B7800):

The Data Reference Unit (DRU) fetches data from associative memory or main memory so it can be used by the EU.

CENTRAL DATA BUFFER (B7800):

The central data buffer (64 words) takes the place of some execution unit queues and the stack buffer. It acts as storage and an exchange between CPM units.

INTERRUPT BUS

Module to module interrupt line used to interrupt other CPM's and IOM's.

The CPM will do internal residue, parity and continuity checking. If errors are detected a CPM fail word is generated and an interrupt occurs. The CPM is also capable of instruction retry.

The CPM must read a register (62) every 8 to 16 seconds. If this is not done a EGG TIMER interrupt in generated. This will prevent the CPM from looping in control state.

Associative Memory (ASM) holds copies of main memory words. ASM is loaded when a reference is made to a word which is not in the ASM. When the CPM needs to fetch one word from main memory it will bring in the 3 words around the word requested. Thus, ASM is loaded in 4 word groups. There is an address array which holds main memory addresses associated with words in ASM. A priority list is maintained for ageing out old data. When a word is written to ASM it is also written to main memory or the store queue (B7800).

B7700 ASM is divided into three main units. These units are the data array, address array and priority array. These units can be described as follows.

```
DATA ARRAY
   64 BLOCKS
      EACH BLOCK 4 GROUPS
         EACH GROUP 4 WORDS
TOTAL 1024 WORDS

ADDRESS ARRAY
   64 BLOCKS
      4 GROUPS PER BLOCK
   EACH ADDRESS POINTS TO BASE OF 4 WORDS

PRIORITY ARRAY
   64 BLOCKS
      EACH BLOCK 6 BITS
```

A diagram of the B7700 ASM is given in figure 1-5 thru 1-7. To find a word in ASM bits [7:6] of the address are used to select a block. Bits [19:12] are compared to all four groups in the address array. If a match is found the group in the data array is known. The proper word can be computed from bits [1:2] of the address.

If the word is not in the ASM it is fetched. In addition, three words around the word are fetched (bits [1:2] are truned off for the main memory request). They will be placed in a open group of a block addressed by bits [7:6]. If there is no open group the oldest group is replaced.

The priority array is used to keep track of access frequency by block. This information is used to determine which group to remove when there is contention in a block.

A READLOCK operation will purge the contents of ASM. There are other operations which will cause ASM to be purged (eg. setting some registers).

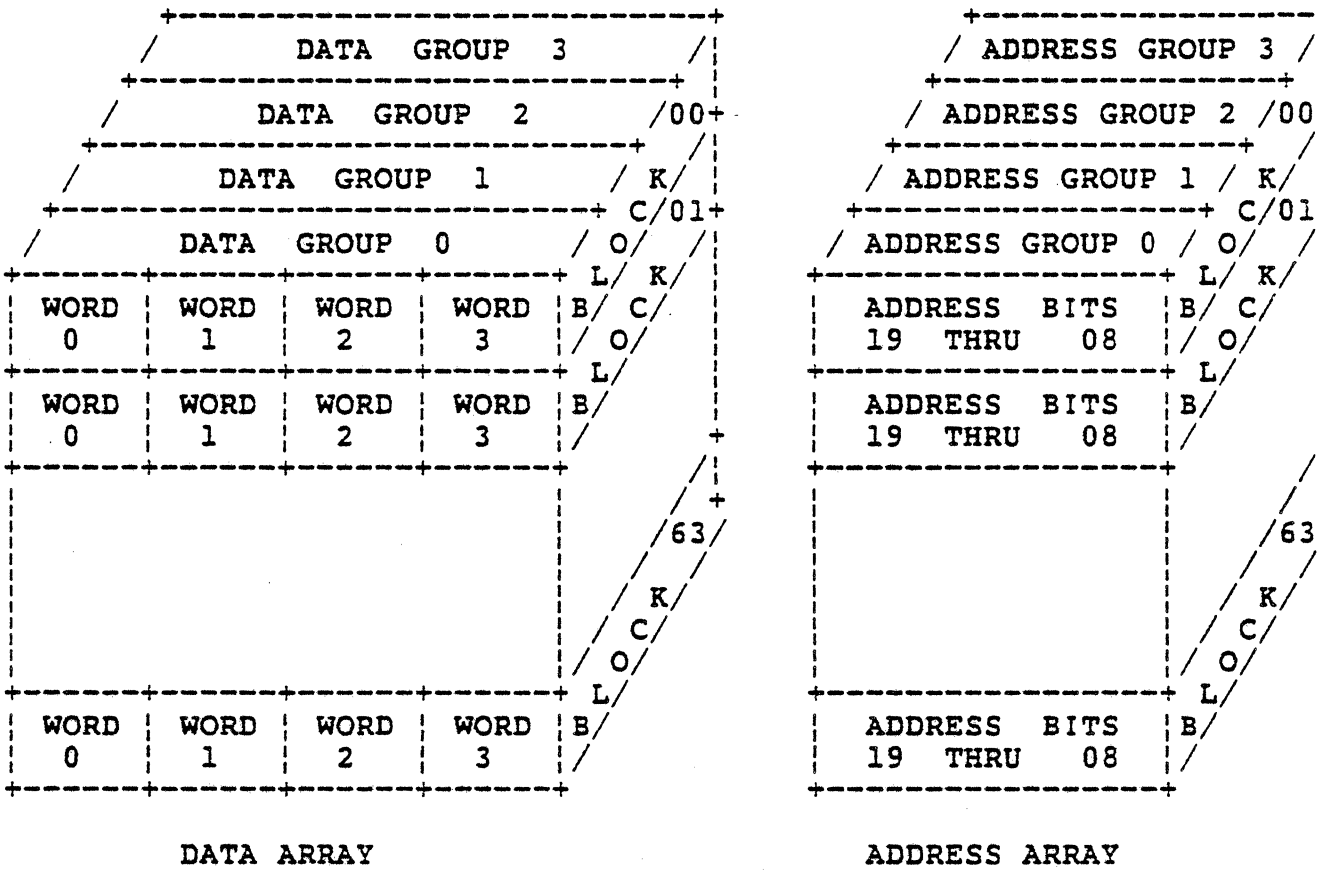


Figure 1-5. B7700 Associative Memory

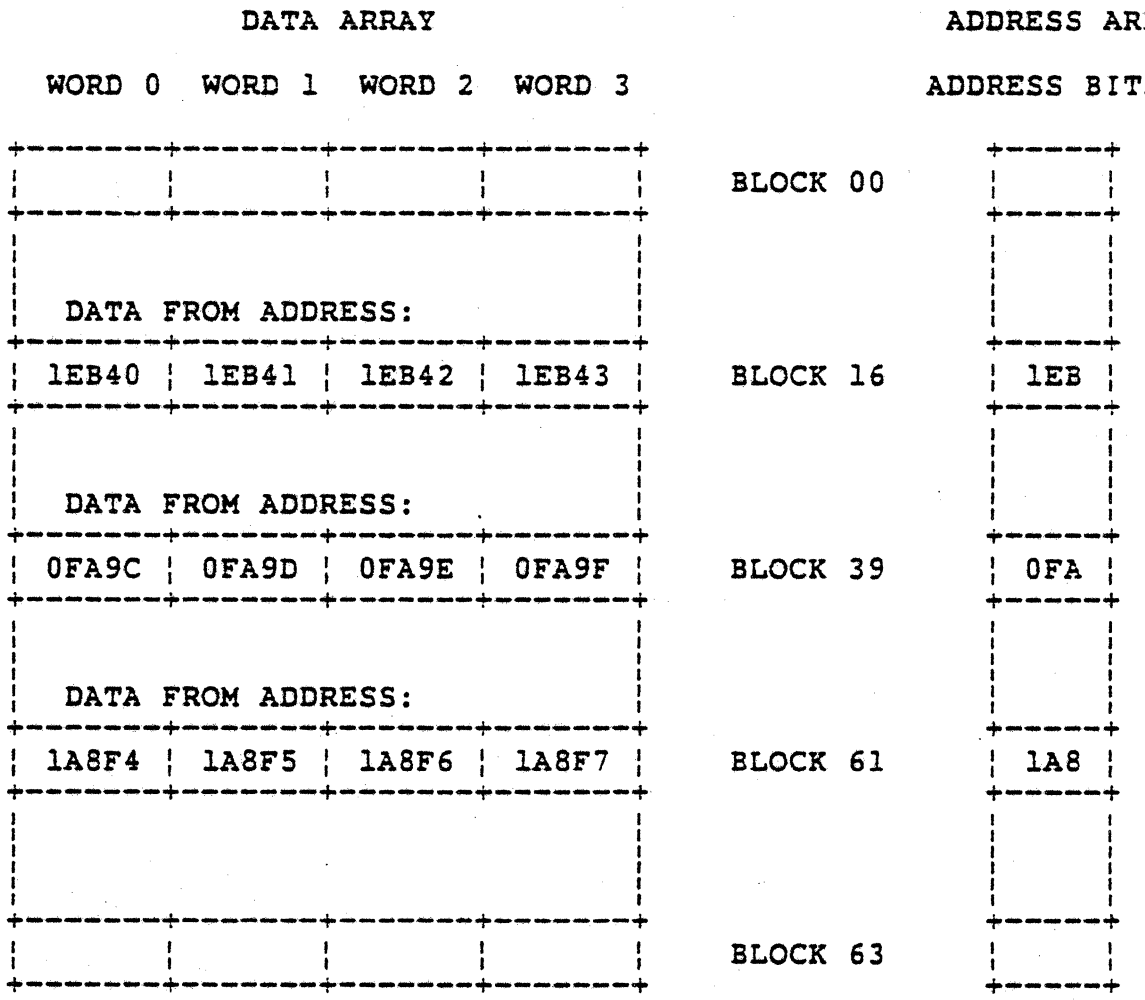


Figure 1-6. B7700 Associative Memory

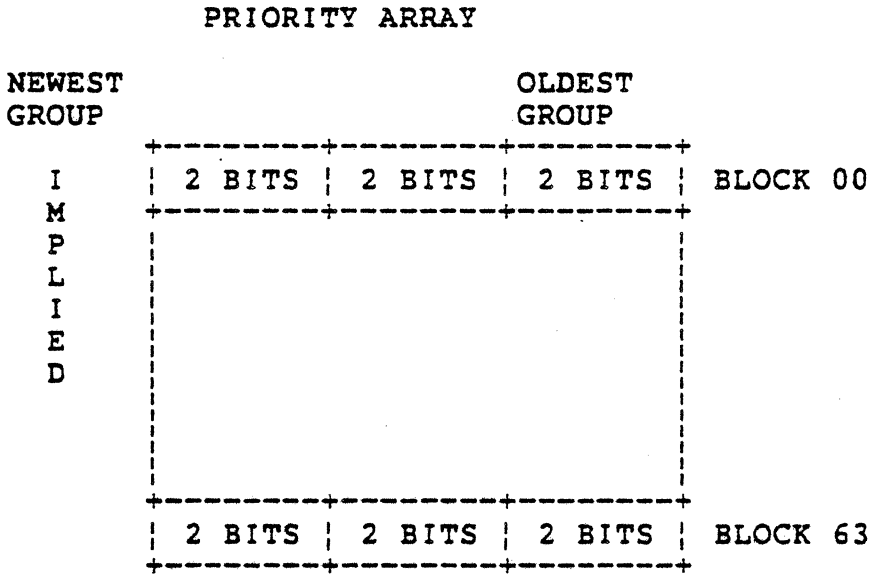


Figure 1-7. B7700 Associative Memory

IOM

Figure 1-8 is a block diagram of a B7000 IOM. Following is a description of the units in the IOM.

MEMORY INTERFACE UNIT

The memory interface unit provides memory interface for the IOM. It buffers and controls all IOM memory access. The memory interface unit controls access priority. The priority is:

DSU - Data Service Unit
PCI
DFIA
DFIB
Lowest number channel in each unit has highest priority
Data comm interface unit
Translator

TRANSLATOR UNIT

The translator is the control unit of the IOM. It fetches commands from memory and starts I/O operations. The translator sends interrupts for I/O finish and ODT status change. It holds 20 bit addresses for Home Address (HA), Unit Table (UT), Queue Head (QH) and Status Queue (SQ). In addition, the translator contains channel status information, peripheral status and controls conditional I/O operations.

DATACOMM INTERFACE UNIT

The datacomm interface unit transfers information between the IOM and DCP. Thus, it provides the memory interface for the DCP.

SCAN INTERFACE UNIT

The scan interface is for scan type operations between DCP's and DFO's connected to the IOM.

DATA SERVICE UNIT

The data service unit is made up of peripheral interface unit, Disk File Interface "A" (DFIA), and Disk File Interface "B" (BFIB). It forms the buffer between memory and peripherals. There are two (2) four word buffers on each channel which allows the IOM to take advantage of 4 word phased access to

memory.

CONNECTED TO DATA SERVICE UNIT

A Peripheral Control Cabinet (PCC) bus is connected between the IOM and PCC. The PCC contains peripheral controls. Data is transfered to the IOM in one or two byte groups. The PCC is independently powered or powered by B6700 style AC mods.

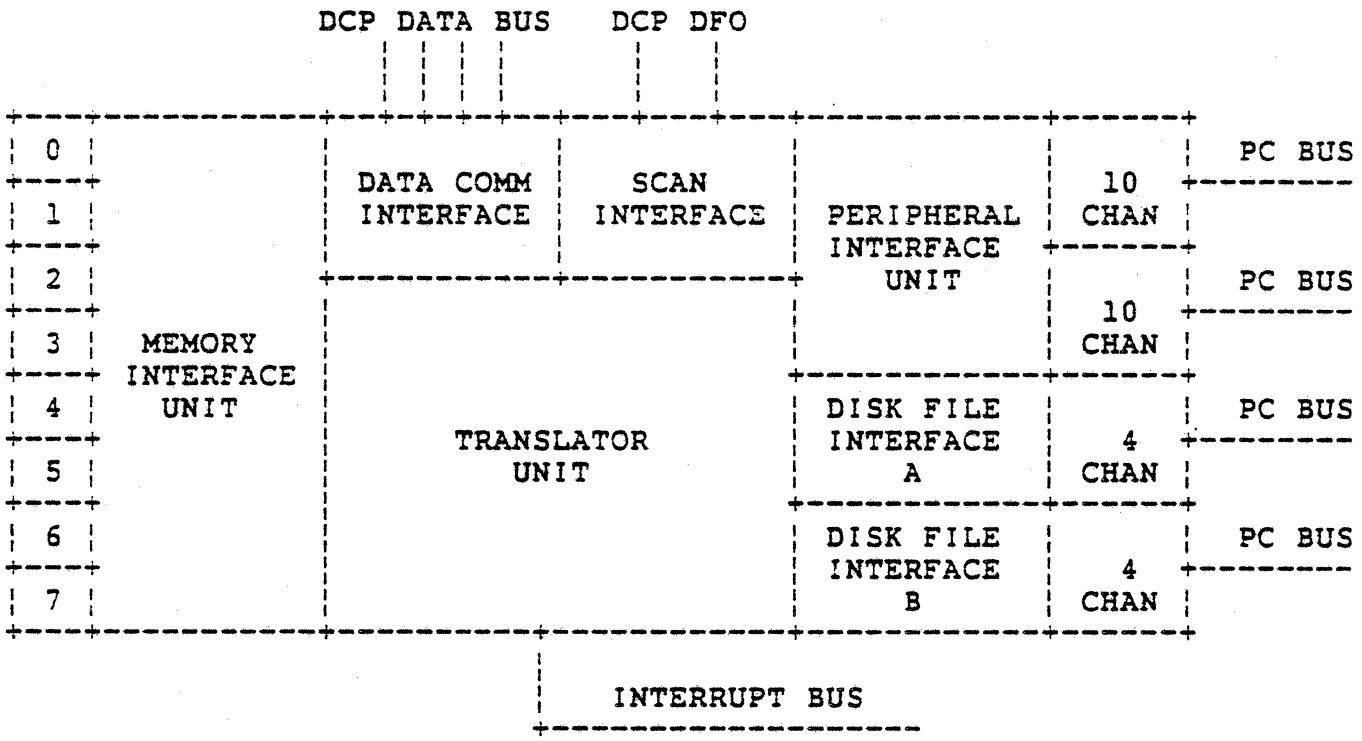


Figure 1-8. IOM Block Diagram

Figure 1-9 is a diagram of the structures used by the IOM and MCP. Following are notes about these structures:

I/O QUEUE:

All I/O'S for a unit are linked to each other. The first I/O Control Block (IOCB) is pointed to by the I/O queue head word. The last IOCB is pointed to by the I/O queue tail word. The linkage from IOCB to IOCB is thru a link word in the IOCB. All IOM's will use the same I/O queue. A queue for a unit is started when the first IOCB is placed in the I/O queue. The IOM will process the next IOCB in the queue without MCP intervention. Thus, a queue is only started when a IOCB is placed in a empty queue.

RESULT QUEUE:

When an I/O has finished, the IOM will place the IOCB in the result queue. Each IOM has it's own result queue. All I/O operations done by an IOM are placed in the same result queue.

The MCP can decide when it wants to be interrupted by the IOM. This interrupt will cause the MCP to look at the result queue. The system can be set up so the IOM will interrupt:

When an I/O queue is empty
On each I/O finish
If a task is waiting on an I/O
If a CPM is idle

UNIT TABLE:

The unit table contains unit information used by the IOM and MCP. This information includes ring walk information, number of channels on an exchange and a lock bit for the unit table and I/O queue.

HOME ADDRESS:

The home address words are used by the MCP to give the IOM commands. The types of commands to an IOM include load registers (HA, UT, QH, SQ), start I/O, interrogate peripheral status and scan out DCP (INITIALIZE, HALT and ATTENTION NEEDED). There are 6 home address words. They are defined as follows:

HA[0]
Command to IOM.

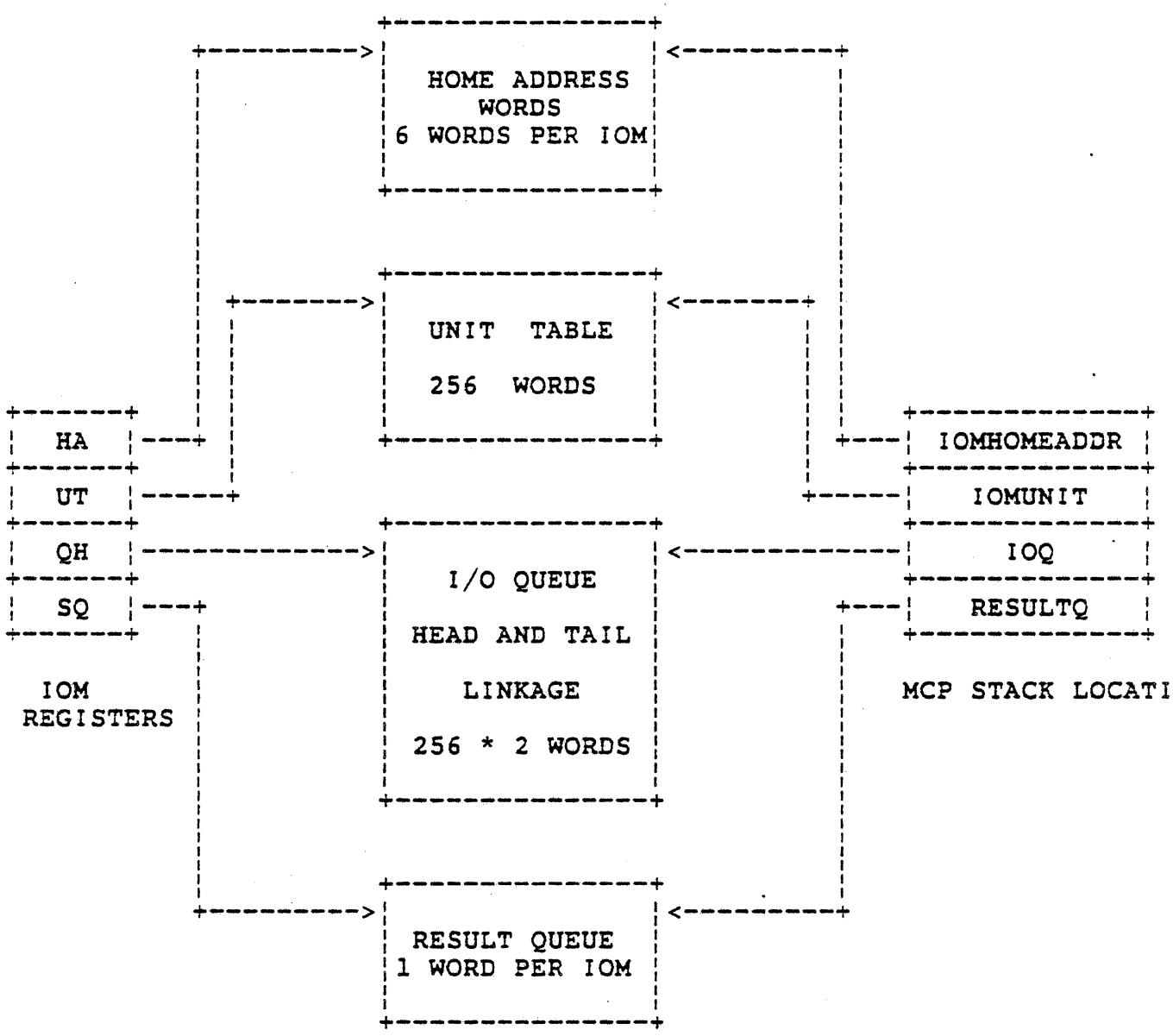
HA[1]
Special information (peripheral status).

HA[2]
HALOCKWORD. Buzzed by MCP to get control of Home Address words.

HA[3]
Not used.

HA[4]
IOM fail word, or hard H/L result descriptor.

HA[5]
Sync I/O result descriptor.



These structures are in memory and can be addressed by the IOM or MCP.

Figure 1-9. IOM Structures

IOCB:

An I/O operation is defined by an IOCB (18 words). The first 7 words are used by the IOM (hardware) and the MCP. The other words are used by the MCP only. The words in an IOCB used by hardware are:

0 IOMNEXTLINK.
Link to next IOCB.
1 IOMSIDELINK.
Not used by hardware.
2 IOMAREADESC.
Buffer descriptor, address and length.
3 IOMIOCW.
IOCW for I/O.
4 IOMCDL.
CDL built by software for IOM.
5 PHYSICALRD.
Hardware result descriptor.
6 IOMTIMECELL.
Channel busy time.

The other words of an IOCB can be found in the MCP. These words include I/O mask, reference to event, reference to FIB and other information.

I/O FLOW

The flow of an IOCB will be traced. This flow will start with the MCP building an IOCB. The required words are placed in the IOCB. The MCP will link the IOCB into the proper I/O queue. The following procedure is used:

```
BUZZ47(IOMUNIT[UNITNO]) % LOCK UNIT TABLE AND I/O QUEUE
GET TAIL WORD FOR UNIT
IF THE TAIL WORD IS 0 THEN
   PLACE ADDRESS OF THIS IOCB IN TAIL AND
    HEAD WORD FOR THE QUEUE
ELSE
   PLACE THE ADDRESS OF THIS IOCB IN THE IOMNEXTLINK
    WORD OF THE LAST IOCB IN THE QUEUE.
   PLACE THE ADDRESS OF THIS IOCB IN THE
    TAIL WORD FOR THE QUEUE.
UNLOCK AND STORE IOMUNIT[UNITNO]
IF THIS I/O IS THE FIRST IOCB IN THE QUEUE THEN
   BUZZ(HALOCKWORD)
   STORE A START I/O COMMAND IN HA[0]
   INTERRUPT THE IOM
   UNLOCK THE HALOCKWORD
```

Before we look at what the IOM does when it is interrupted a few comments need to be made about

the start I/O process.

Bit 47 of a home address command is called the lock bit. This bit must be on for all IOM commands.

All IOM's that have a path to the unit will be interrupted. Only one IOM will do the I/O operation. The MCP maintains information of which IOM's to interrupt for a given unit.

The IOM will read HA[0] then zero it. Before a CPM places a new command in HA[0] it will make sure the IOM has cleared the last command. If the last command has not been cleared the CPM will wait .75 sec. If the command is still not clear the CPM will interrupt the IOM again. If the command is still not clear after .75 sec the IOM will be removed from the system.

Once ring walk is entered the IOM will follow the next unit field of the unit table word looking for a unit with the JB bit on. This bit indicates the I/O was not done because there was no channel available at the time. Ring walk will give all units on an exchange equal priority.

The IOM has a thruput scheduler for I/O's which are marked by the MCP (DISK, PACK and TAPE). A count of the I/O's in process will be kept and compared against a maximum value (set by F.E.). When the maximum value is reached the IOM will queue requests to start more I/O's. This will only be done for units in the PCI. The queue size for start I/O commands is 16. If the queue is exceeded, the IOM will not respond to a start I/O command.

Fail IOCB's will be generated by the IOM if an error is detected by the IOM which is not associated with a unit. The IOM will use IOCB's in unit 0. It will place a fail word in the PHYSICALRD word of the IOCB and place it in the result queue. The CPM will be interrupted. The IOM will stop processing.

An I/O error to a unit will stop processing on that unit.

Figure 1-10 thru 1-14 are flow charts of start I/O and finish I/O operations.

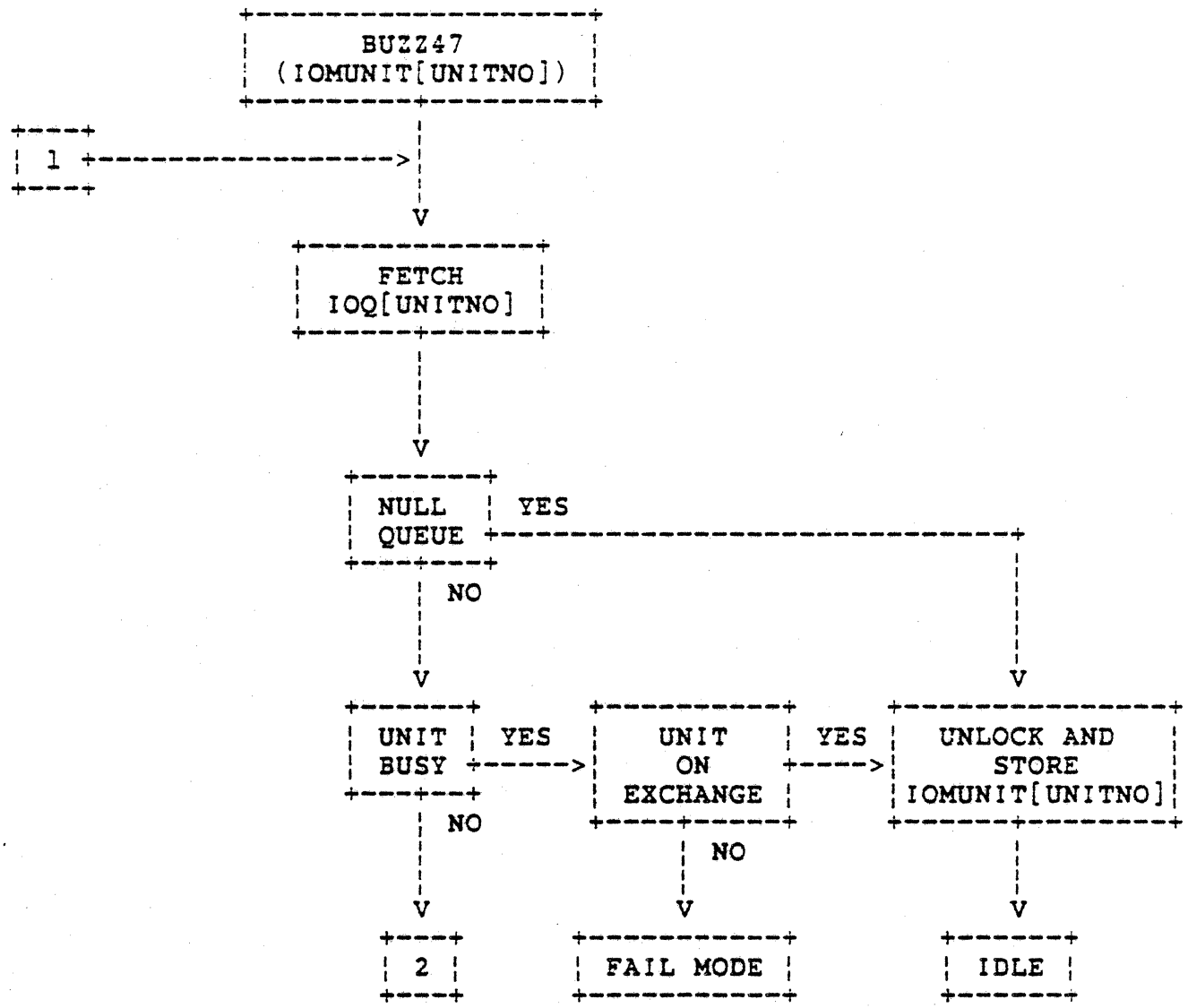


FIGURE 1-10. START I/O (PAGE 1 OF 2)

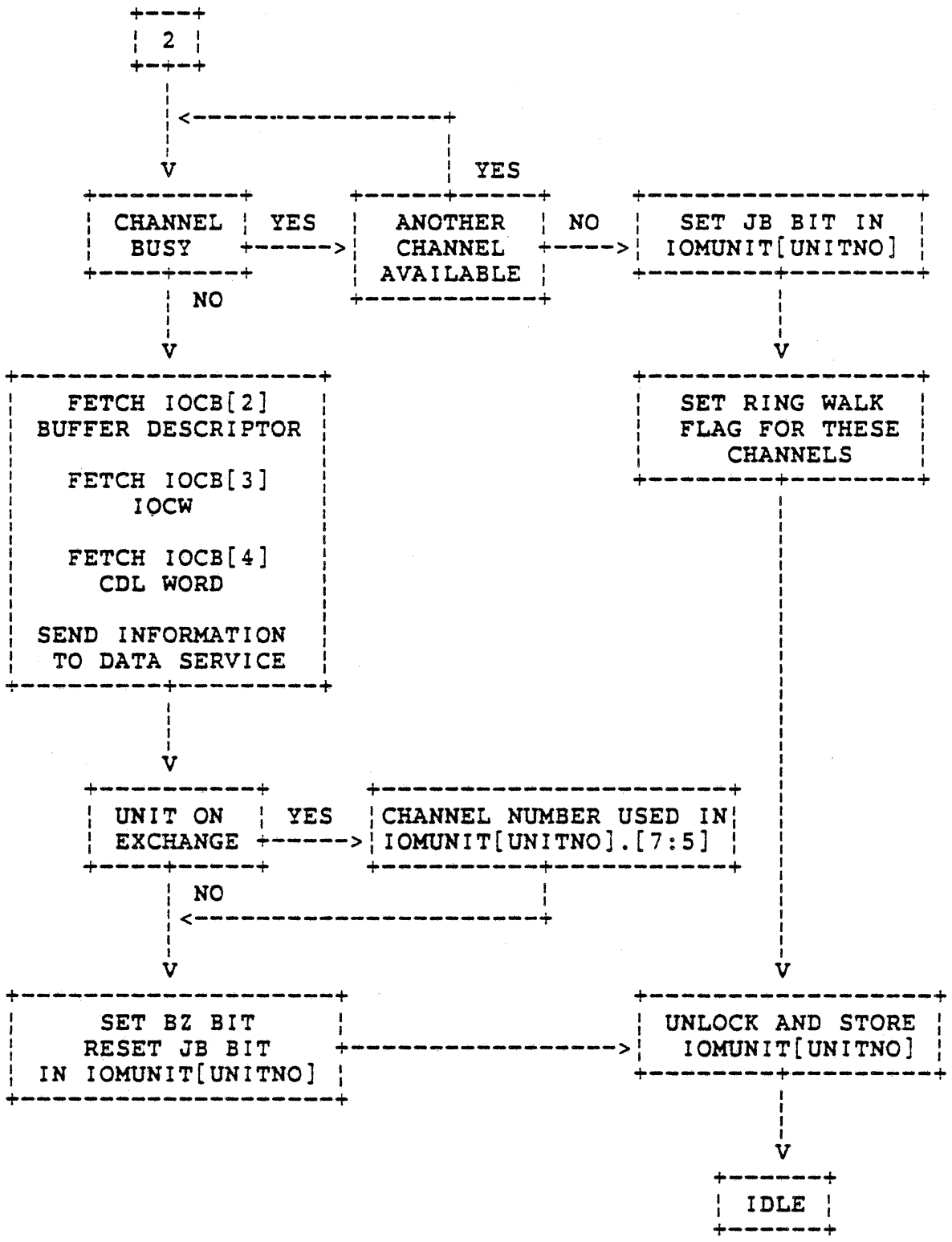


FIGURE 1-11. START I/O (PAGE 2 OF 2)

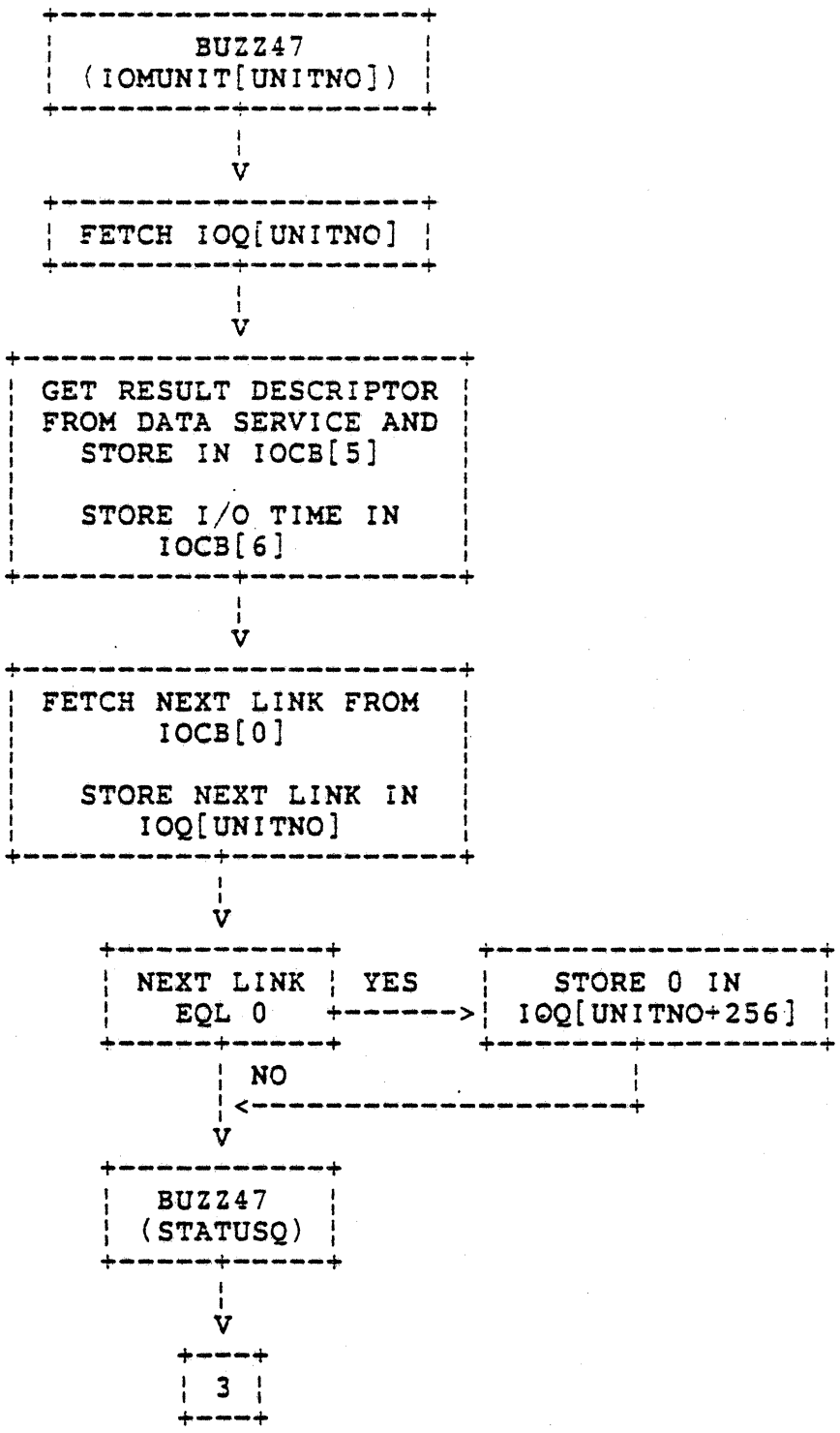


FIGURE 1-12. TERMINATE I/O (PAGE 1 OF 3)

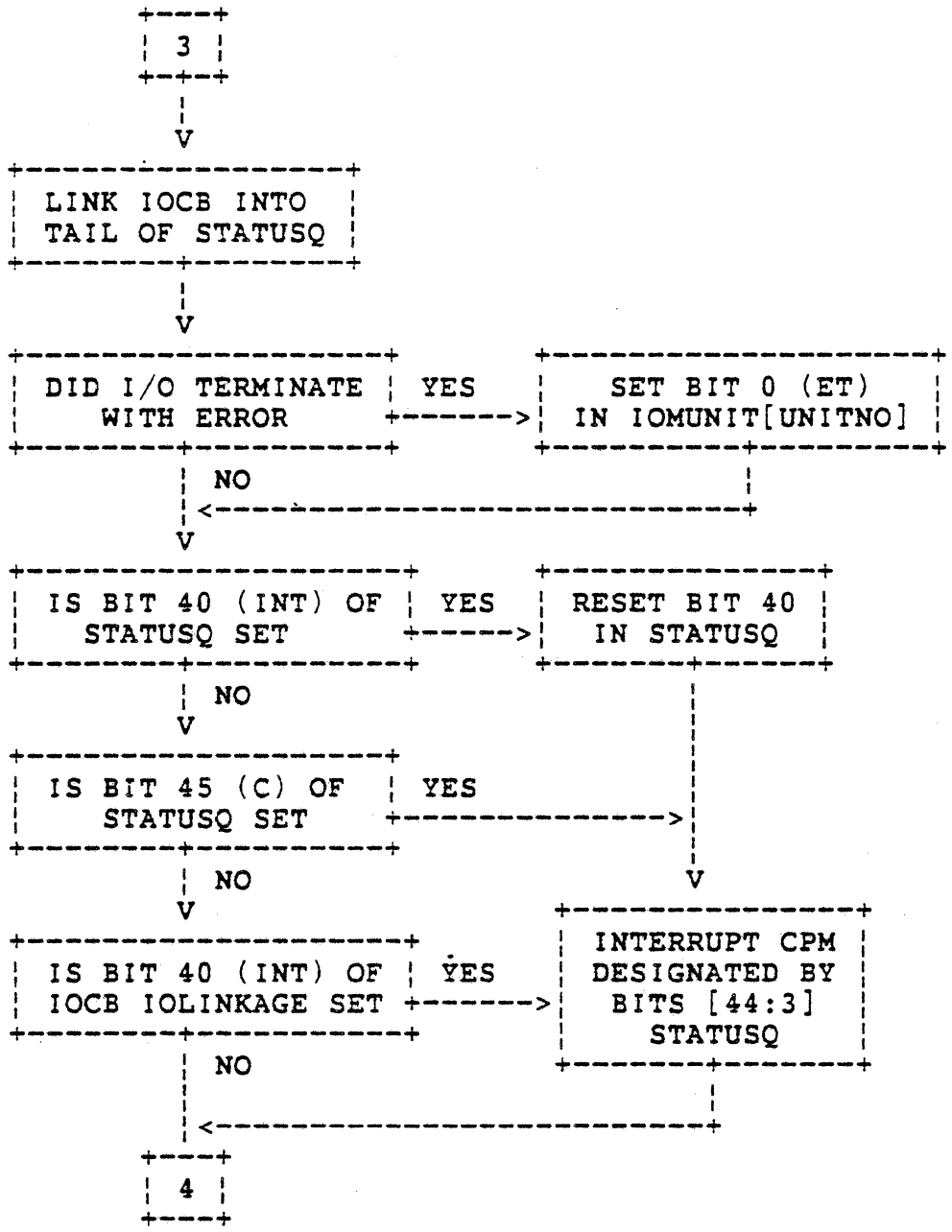


FIGURE 1-13. TERMINATE I/O (PAGE 2 OF 3)

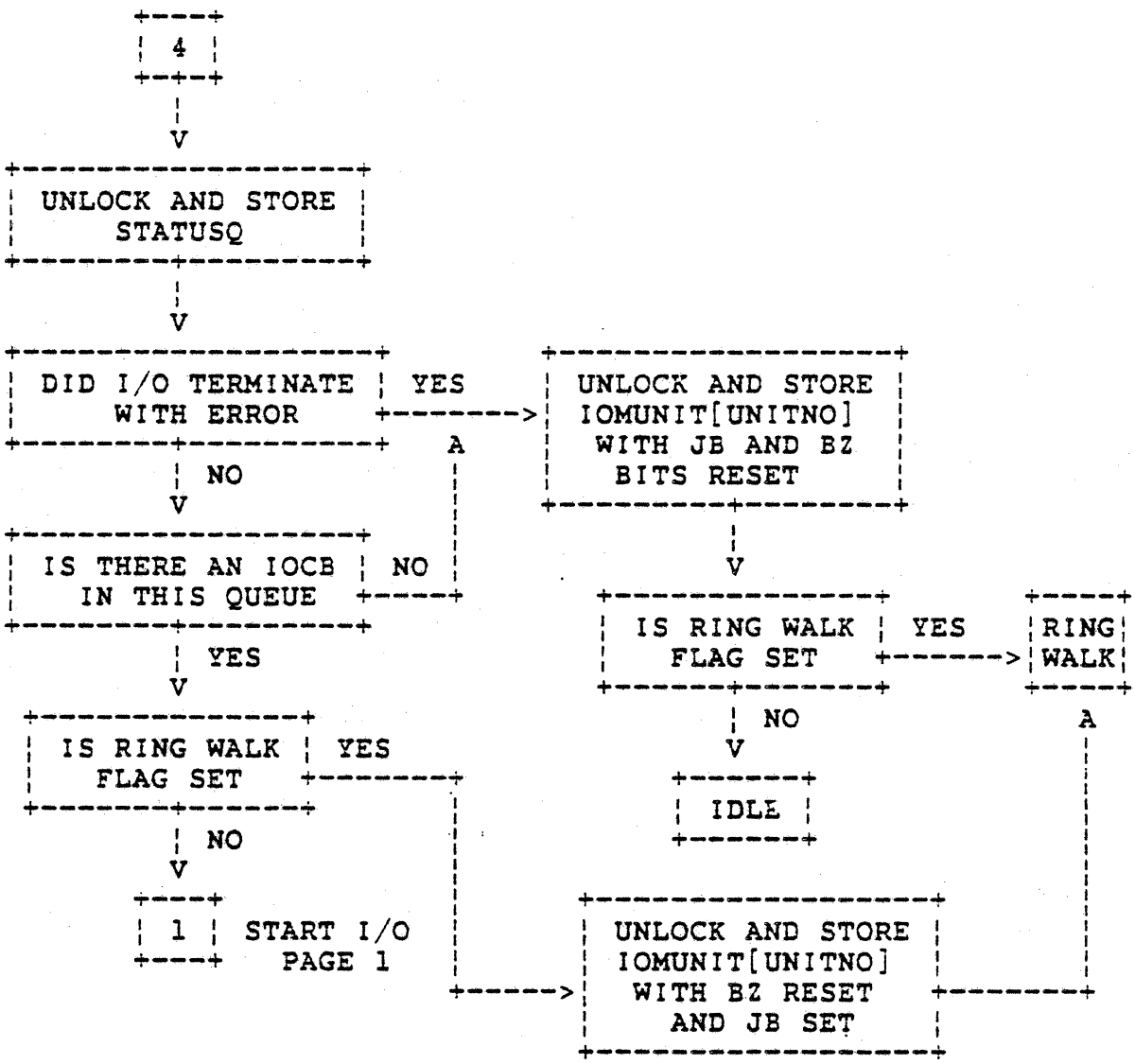


FIGURE 1-14. TERMINATE I/O (PAGE 3 OF 3)

SYSTEM INITIALIZATION

System initialization begins when the operator presses the enable and load buttons on the operator control console (figure 1-15). The hardware takes over the initialization process when the operator releases the enable button and then the MCP continues this process by setting up the disk directory or verifying the quality of an existing directory, analyzing the system quality and configuration and, in general, bringing the system up to the state necessary to accept job input either via ODT's or card readers, etc.

Figure 1-15. Operator's Control Console

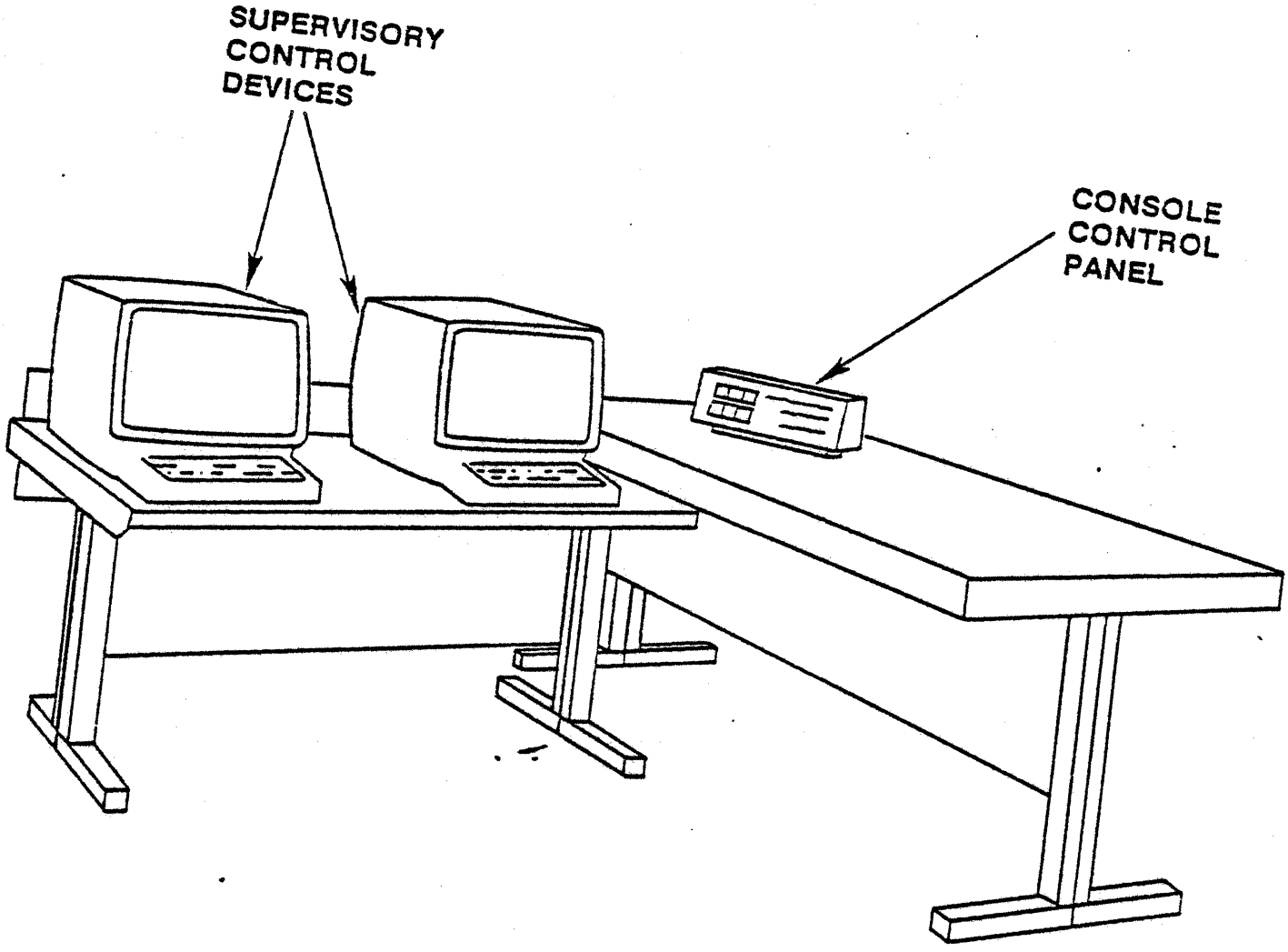


Figure 1-15. Operator's Control Console

The COLD START/HALT LOAD SELECTION CARD (figure 1-16) plays an important role in the hardware part of the initialization process. Each CPM and IOM will have one of these cards mounted in its backplane. Of particular interest on this card is switch 7, the A/B switch. This switch indicates which partition the module is in and switch 8 is up for the IOM selected to take part in the initialization within that partition. Switches 14, 15 and 16 must be set to give the box number of the processor that will take part in the process. Several other switches are important on this card but will not be discussed at this time.

Figure 1-16. Cold Start/Halt Load Selection Card

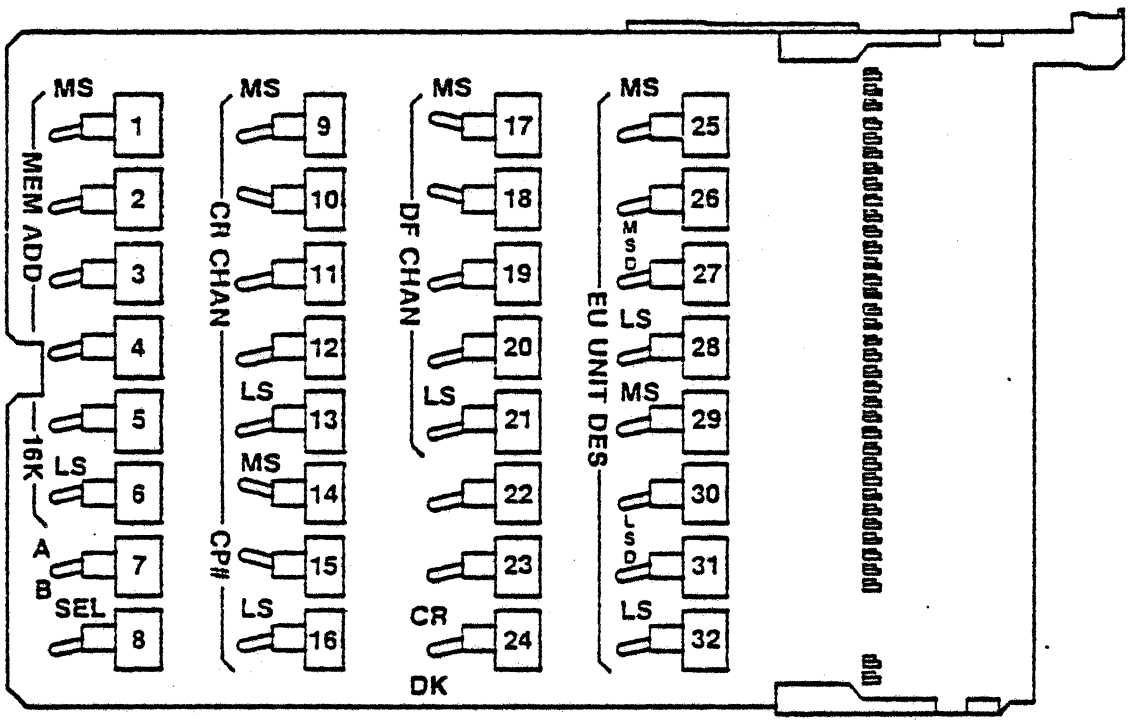


Figure 1-16. Cold Start/Halt Load Selection Card

System initialization consists of a series of bootstraps as shown in figure 1-17. Depending on the state of the operator's console, either the card load or the disk path will be taken. Regardless of the path taken, the MCP will end up in control of the system and prepared for work. Figure 1-18 shows the format of the three most important words used by the system during the hardware and first level software portion of figure 1-17. Figure 1-21 shows the basic central memory set up after initialization is complete.

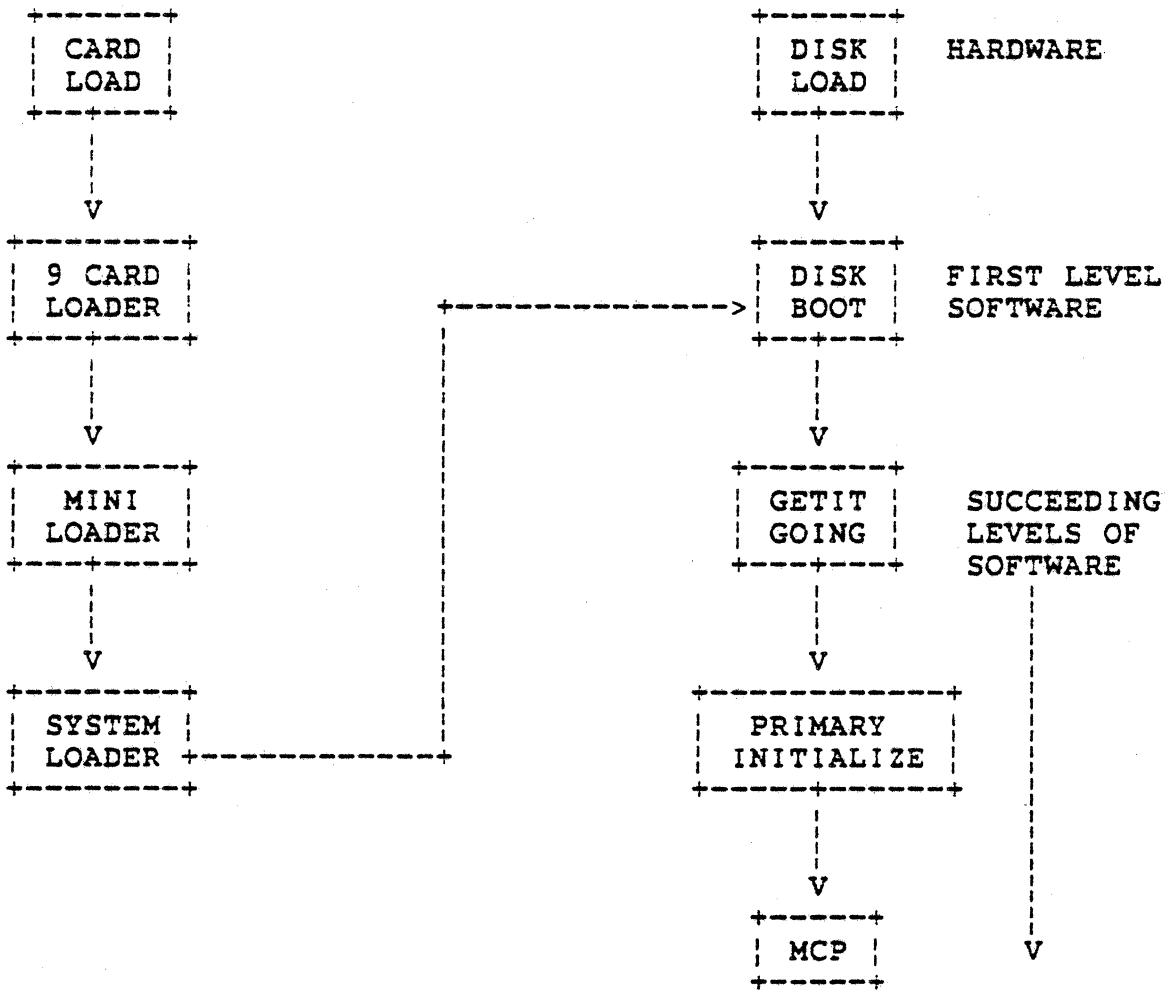
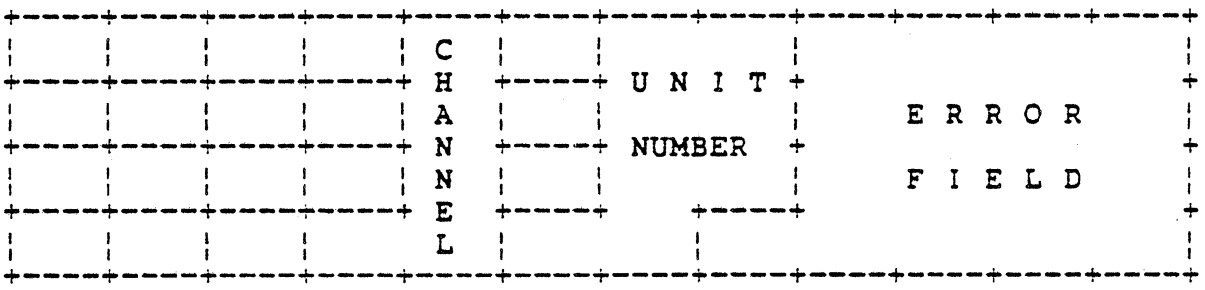
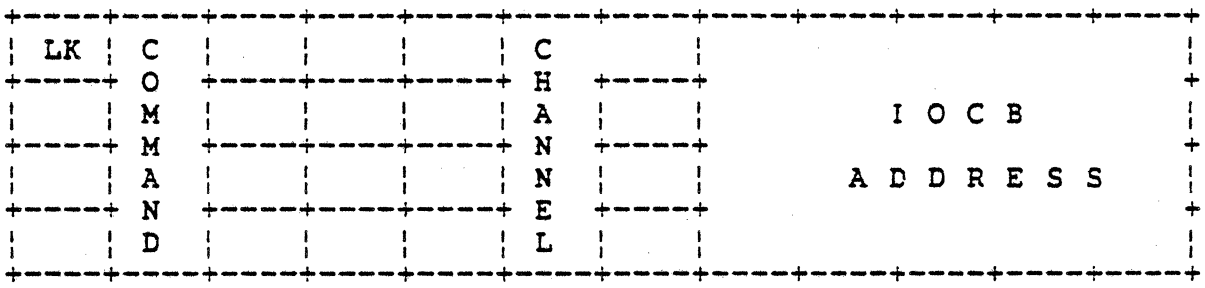


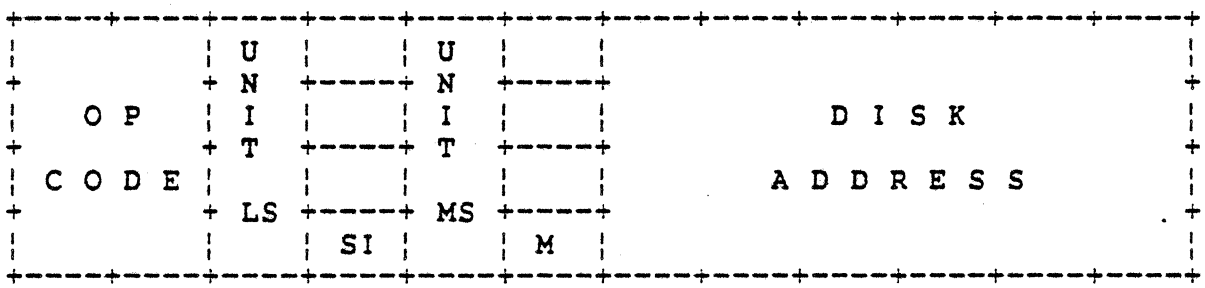
Figure 1-17. Bootstrap Organization



HARDLOAD RESULT DESCRIPTOR AT MEMORY[4]



SYNC I/O COMMAND WORD AT HOMEADDRESS[0]



DISK CDL AT IOCB[4]

Figure 1-18. Disk Boot Words

WORK FLOW MANAGEMENT

Once the system has been initialized, the MCP is ready to allow jobs into the mix and to perform work requested by these jobs. The Work Flow Management part of the MCP is that section used to control the introduction of jobs into the system. One of the objectives in designing the Work Flow system was to localize those functions relating to work flow control so as to facilitate construction of installation supervisor programs. The MCP has been organized in order to meet this objective. In particular, all scheduling and operator interface functions have been localized in a program unit called the CONTROLLER.

The WFM operating system includes four separately compiled program units: the MCP, the CONTROLLER, the WORK FLOW LANGUAGE (WFL) COMPILER, and the JOBFORMATTER. All but the MCP which is written in NEWP are written in DCALGOL and are bound into the MCP. Another section, not part of the WFM system but also bound into the MCPHOST is the ALGOLSUPPLEMENT. The CONTROLLER manages the display routines and the scheduling queues provided by WFM. The WFL compiler handles the control cards necessary for job control. JOBFORMATTER provides for the printing of job summary information as an adjunct to printing backup files.

Figure 1-19 shows how these separate units interact. There are two external influences on the system: job decks and operator input. Subsequent sections in the manual document how each is handled by the system.

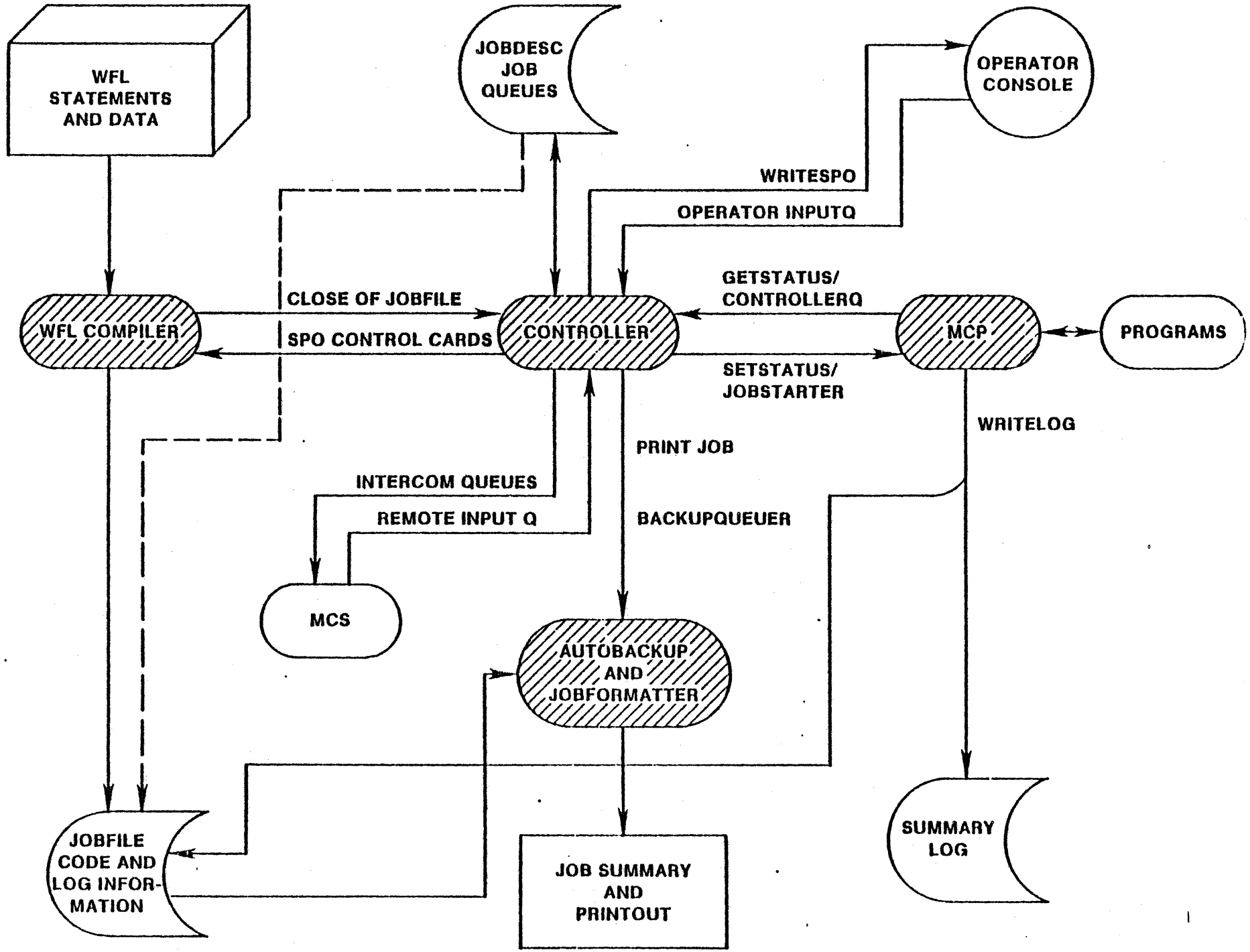

WFL
STATEMENTS
AND DATA
JOBDESC
JOB
QUEUES
OPERATOR
CONSOLE
WRITESPO
OPERATOR INPUTQ
GETSTATUS/
CONTROLLERQ
CLOSE OF JOBFILE
SPO CONTROL CARDS
WFL COMPILER
CONTROLLER
MCP
PROGRAMS
SETSTATUS/
JOBSTARTER
WRITELOG
PRINT JOB
INTERCOM QUEUES
REMOTE INPUT Q
BACKUPQUEUER
MCS
AUTOBACKUP
AND
JOBFORMATTER
JOBFILE
CODE AND
LOG INFOR-
MATION
JOB SUMMARY
AND
PRINTOUT
SUMMARY
LOG

Figure 1-19. WFM System Organization

WFL Compiler

When a job deck is placed in the card reader, the ETERNALIR procedure of the MCP calls up the WFL compiler as an invisible independent runner. (The WFL compiler can also be programmatically invoked.) The WFL compiler checks syntax and translates job control statements into machine code. The WFL compiler also creates a special type of file called a jobfile for each job inserted in the card reader or input through a ODT. It places into this file object code to run the tasks of that job, copies of the control cards (for later printout), data decks belonging to the job, and space for logging and restart information. Once compiled, a jobfile is a self-sufficient representation of a job. When the compiler closes a jobfile, the CLOSE routine of the MCP notices that this is a jobfile. Rather than entering the file in the disk directory, CLOSE leaves it in the DISKFILEHEADERS stack and inserts a message into the CONTROLLERQ (SCHEDULEREQUEST) indicating that a new job has entered the system and specifying its index in the DISKFILEHEADERS stack.

CONTROLLER Routine

When a job's control cards have been analyzed, it is given to the CONTROLLER, which is fired up as an independent runner at Halt/Load time. The CONTROLLER is in charge of queue-level scheduling and operator communication. It receives notice of operator inputs and certain other system events via input queues. The CONTROLLER can request, in turn, any useful aspect of the system status and change that status, for example: purge tapes, DS or ST jobs, or MC programs.

Queue-Level Scheduling

Queue-Level scheduling is more efficient than MCP-level scheduling in two important aspects. First, queue-level scheduling absorbs less system resources than MCP-level scheduling. It thus provides a practical way to make large numbers of jobs visible candidates for system or operator selection. Second, an installation may define multiple queues for different classes of service. It could, for example, set up one batch-queue and one quick-service queue. Jobs from both queues would be visible when the system wanted to start a new job. Since turnaround of a job in a queue is related to the time required by the longest job in the queue, multiple queues can improve service for short jobs.

The CONTROLLER maintains a Jobfile Description File (JOBDESC) which has the dual roles of directory for the jobfile disk areas and queue for the CONTROLLER's scheduling. Entries in it consist of the file headers for the jobfiles, plus links used by the CONTROLLER to organize the jobs by class and priority. The CONTROLLER procedure ABSTRACT chooses the

appropriate class by matching requirements of the job (inserted by WFL into the jobfile) with the specifications of the various queues. The broken line in figure 1-19 illustrates that the job queues contain pointers to the various jobfiles, ordering them by queue and priority.

The CONTROLLER routes jobs to appropriate queues. This may be specified explicitly by a CLASS specification. When a job contains no such statement, it is put into the default queue, if one exists, and the job has no resource restriction statements.

The system provides one queue by default: queue 0. This queue has the lowest priority in scheduling, since the queue number has an implicit priority on the queue. The installation may modify or delete queue 0 as well as add others. It should be noted, however, that no jobs will be run if there are no queues. All jobs would be terminated with the error message JOB DS-ED OUT OF QUEUE, which means there is no acceptable queue for them to join.

When a job has resource restrictions specified, it is put in the highest queue whose limits it does not exceed. The job is then subjected to two stages of scheduling with different selection rules and resource demands. While it is in the scheduling queue, its selectability depends on the queue number and queue attributes called MIXLIMIT and TURNAROUND.

Eventually the Controller will select the job and present it to the MCP for processing. At that point, the standard MCP scheduling algorithms based on core estimates and priority take over.

The limits associated with a queue describe the maximum resources which a job from that queue may use. The possible limits are on priority, I/O time, process time, subspaces, tapes allowed (i.e., resource statement), lines printed, cards punched, etc. If a job has, for example, a MAXPROCTIME control statement, then that statement must indicate less time than the queue's limit to be allowed in that queue. If no limit is specified for the queue, all jobs are acceptable with respect to that attribute. If a job is not acceptable to any queue, it is terminated with a JOB DS-ED OUT OF QUEUE error message. If the job gave no restriction, the job can go into any queue; but it is then assigned default restrictions from the queue it enters. (This is valid only if the compile time option QFACTMATCHING is set.) See figure 1-20 for a more complete version of the queue-matching algorithm.

Figure 1-20. Job Enqueuing Algorithm

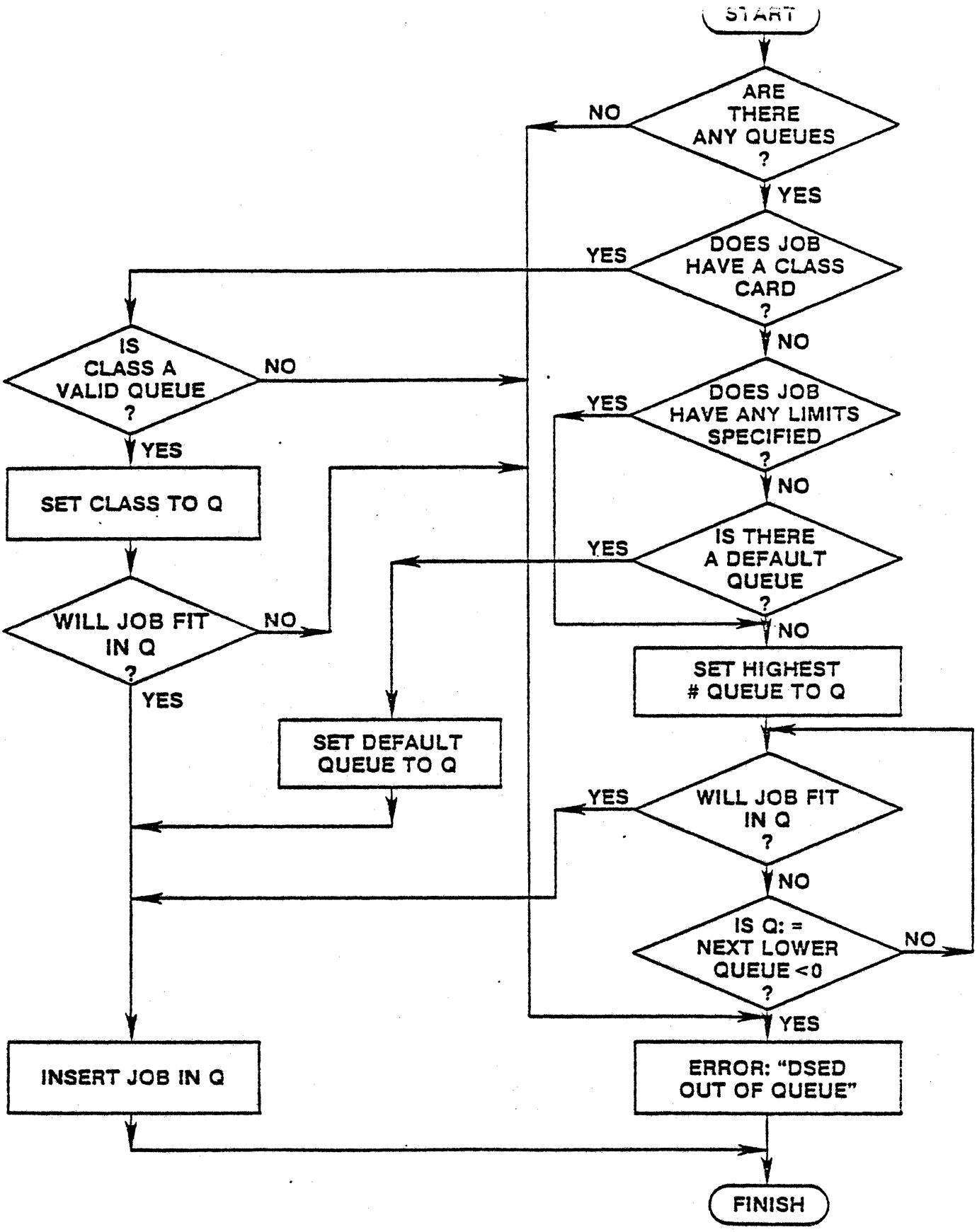


Figure 1-20. Job Enqueuing Algorithm

If an attribute is specified as a limit, only jobs which have a lower limit can enter the created queue. If the attribute is specified as a default, any task without an explicit limit on that attribute is assigned the queue default. These defaults are applied at the time the job to be run is selected from the queue, not at the time of insertion into the queue. The limit on PRIORITY applies to operator input for a job in a given queue as well as to jobs being enqueued. The operator may not change a job's priority higher than the limit of the queue it is in.

While the job is in the queue, the job can be cancelled, have its priority changed, or it can be explicitly selected. Also, the queues can be interrogated, changed, or otherwise have their attributes manipulated.

If a queue is altered or eliminated, all jobs waiting to be selected from it are reexamined to see if they can still go in that queue or, if not, can be redistributed to other queues. Once a job has been selected and given to the MCP, however, it is not affected by queue changes unless there is a Halt/Load. A Halt/Load requeues aborted jobs for restart; therefore an intervening MQ or EQ ODT message may affect those jobs.

CONTROLLER-MCP Interface

The CONTROLLER procedure SELECTION chooses a job to be started paying due attention to the queue priorities and mix limits. Using the queue default attributes, it assigns any job attributes which were not explicitly set by the job. The Controller then calls the MCP procedure JOBSTARTER, passing it the job entry. The job is removed from the queue.

JOBSTARTER transfers the job entry into the proper task/stack structure to run. It enters the jobfile header back into DISKFILEHEADERS. Then it initiates the task via DOCTOR.

While the job is active (or scheduled at the MCP level), its jobfile is used for several things. First, it is the code file for job control statements, so that it is referenced, as any code file, to load executable code into memory. Second, it contains the card data input needed for the job plus pointers to that data. Third, it is the repository of job related log entries, which, with the control cards, are printed at end-of-job. The logging procedure WRITELOG makes log entries to both the jobfile and the system log (figure 1-19). Finally the jobfile is used as a roll-out space between complete tasks, so that the job may be properly restarted in the event of a Halt/Load.

At EOJ, the MCP puts an EOJ notice into CONTROLLERQ. The Controller then enqueues the jobfile for printing; this is done by adding the jobfile index in the length field of the BD queue word it is building. Thus, jobfiles are queued for

printing, just as any BD file would be. When AUTOBACKUP (BACKUPQUEUER) extracts an entry from the BD queue, the length field of the second word is examined for non-zero. If it is non-zero, JOBFORMATTER is called to print the jobfile's control cards and log entries.

Following printing of the jobfile, AUTOBACKUP continues to print any backup printer files for the job. It will also insert an end-of-printing notice into the CONTROLLERQ, so that the CONTROLLER can deallocate the disk space assigned to the jobfile. Some jobs, because of a WFL syntax error or queue limit error, will never be run. These jobs are passed directly from the Controller to AUTOBACKUP without being run or inserted into a queue.

In order to communicate with the MCP, the CONTROLLER calls two routines, GETSTATUS and SETSTATUS. The former returns answers to any interrogations made by the operator; the latter, for example, performs operator requests, including purging tapes, DS-ing jobs, and changing the time. All syntax analysis is done in the CONTROLLER; the requests being given to GETSTATUS and SETSTATUS are in a coded form.

SETSTATUS and GETSTATUS are available to any privileged DCALGOL program. However, the Controller can also get unsolicited information from the MCP via four queues. One queue, the CONTROLLERQ, receives notices about change of state of the system. This includes information used in job scheduling, as well as events needed when a terminal has requested event mode ADM. The second queue is OPERATORINPUTQ. Into this queue KEYIN places ODT inputs. The third queue, MESSAGEDISPLAYQ, receives only display and RSVP messages. The fourth queue, REMOTEINPUTQ, receives all MCS input such as operator input from RJE terminals, and various CANDE inputs. To write out to single line control devices, the CONTROLLER uses another DCALGOL procedure: WRITESPO. Since queues and job level scheduling are managed by the CONTROLLER, it does not need to use SETSTATUS or GETSTATUS to respond to queue related operator inputs, but rather can handle these without MCP intervention, until such point as it must run or print a specific job.

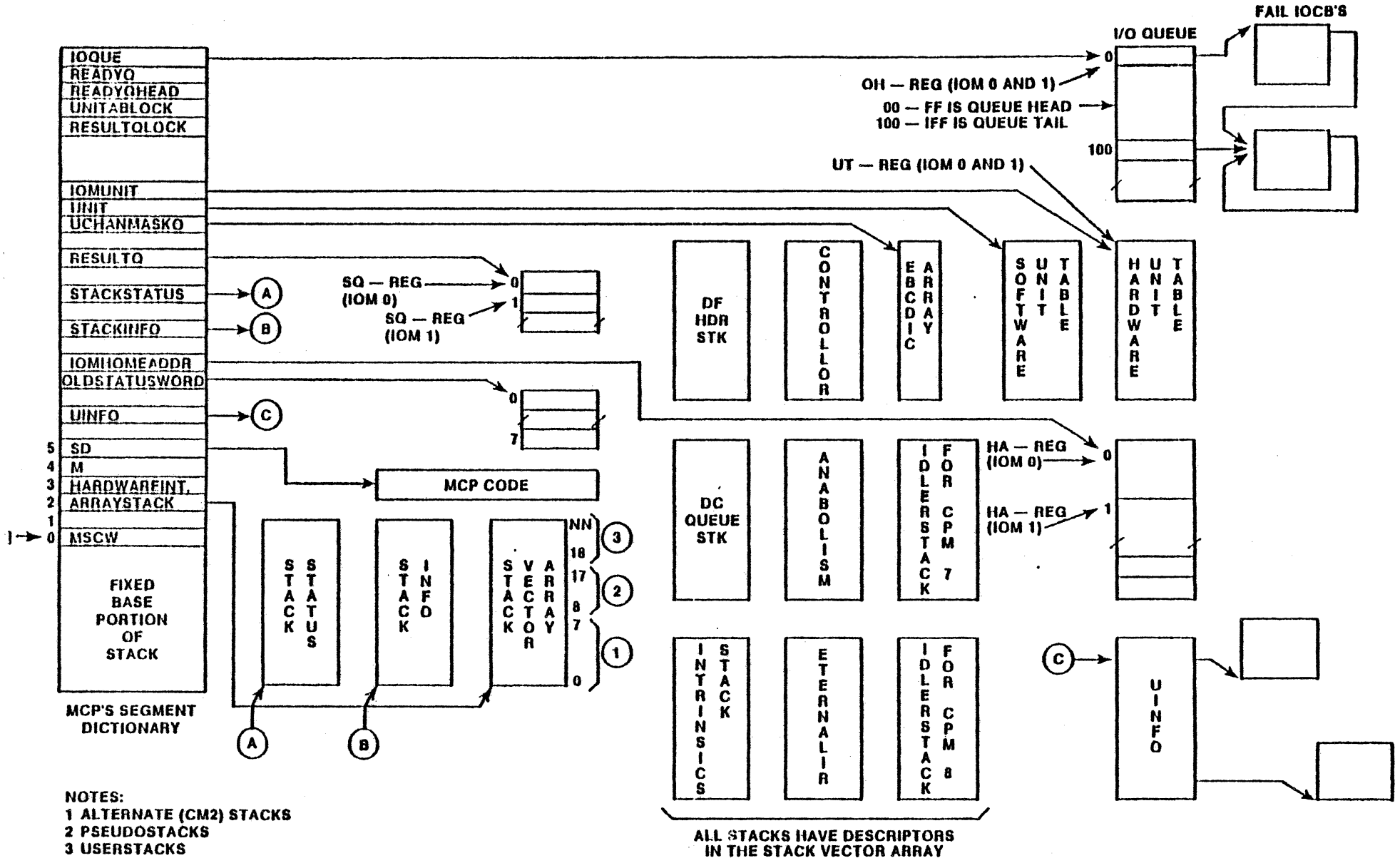

IOQUE
READYQ
READYQHEAD
UNITABLOCK
RESULTOLOCK
IOMUNIT
UINIT
UCHANMASKO
RESULTQ
STACKSTATUS
STACKINFO
IOMHOMEADDR
OLDSTATUSWORD
UINFO
SD
M
HARDWAREINT.
ARRAYSTACK
MSCW
FIXED BASE PORTION OF STACK
MCP'S SEGMENT DICTIONARY
A
B
C
SQ — REG (IOM 0)
SQ — REG (IOM 1)
MCP CODE
STACK STATUS
STACK INFO
STACK VECTOR ARRAY
NN
18
17
8
7
0
3
2
1
I/O QUEUE
FAIL IOCB'S
OH — REG (IOM 0 AND 1)
00 — FF IS QUEUE HEAD
100 — IFF IS QUEUE TAIL
UT — REG (IOM 0 AND 1)
DF HDR STK
CONTROLLOR
EBCDIC ARRAY
SOFTWARE UNIT TABLE
HARDWARE UNIT TABLE
DC QUEUE STK
ANABOLISM
IDLERSTACK FOR CPM 7
HA — REG (IOM 0)
HA — REG (IOM 1)
INTRINSICS STACK
ETERNALIR
IDLERSTACK FOR CPM 8
UINFO
ALL STACKS HAVE DESCRIPTORS IN THE STACK VECTOR ARRAY
NOTES:
1 ALTERNATE (CM2) STACKS
2 PSEUDOSTACKS
3 USERSTACKS

Figure 1-21. MCP Memory Areas

If the operator types in a job control deck at the console (which is possible except for data set statements), the CONTROLLER processes the WFL compiler to parse the statement and create a jobfile in the normal manner.

The CONTROLLER maintains the terminal configuration specifications on disk, at the beginning of JOBDESC. That is, if the operator types in an ADM or TERM message, the newly effective instructions are saved so that they may be recovered after a Halt/Load.

Logging

The MCP procedure WRITELOG provides the access mechanism to enter records in the SUMLOG or a jobfile. It breaks the array passed to it into fixed length records and time stamps them, then sends them to the SUMLOG, and, if the other parameters and stack status permit, to the jobfile. WRITELOG suppresses logging to either of these destinations, based on MCP arrays: SUMLOGOMIT for the system log and JOBLOGOMIT for the jobfiles. When a log request is made, WRITELOG indexes into the appropriate array by the log major type and selects a bit, using the log minor type. If that bit is on, logging is suppressed on the respective file. If the major type is greater than the array length or the minor type greater than 47, the entry is logged: By default, open and close records are suppressed from the jobfiles, as is aborted history from the SUMLOG.

A log entry type has been provided for installation use. To get records of this type into the log, an installation must add appropriate WRITELOG calls to the MCP. (The first four words of these records must still agree in format with the standard entry types.)

ALGOL CODE FILES

When an ALGOL program is compiled, the compiler produces a code file similar to that shown in figure 1-22. The minimum contents of any code file is segment dictionary and object code segments. These can be seen in figure 1-22 along with other information in the file.

Due to various considerations, all code files have a fixed format SEG 0. This disk segment is the first segment in the file and contains a word used by the MCP to locate the segment dictionary. There are descriptors in the segment dictionary that are used to locate the object code. The last segment in a code file is the program parameter block. This segment contains information about the program such as priority, printlimit, process time, etc. and is also located by a word in segment 0 of the file.

SECTION 2.

NEWP

INTRODUCTION

NEWP is a high level programming language similar to ALGOL but with constructs that bring it much closer to the hardware level. These constructs allow main frame modules to interrupt each other, memory limits to be changed, overwrites to be performed anywhere in memory, etc. It should be evident from these special features why only one NEWP program is allowed to run at any given time. That one program, by the way, is usually referred to as the MCP.

This section is composed of 3.1 and 3.2 D-NOTES from the B6000 and B7000. Some comments have been added to these D-notes. At the end of the D-notes are some problems a NEWP programmer could encounter.

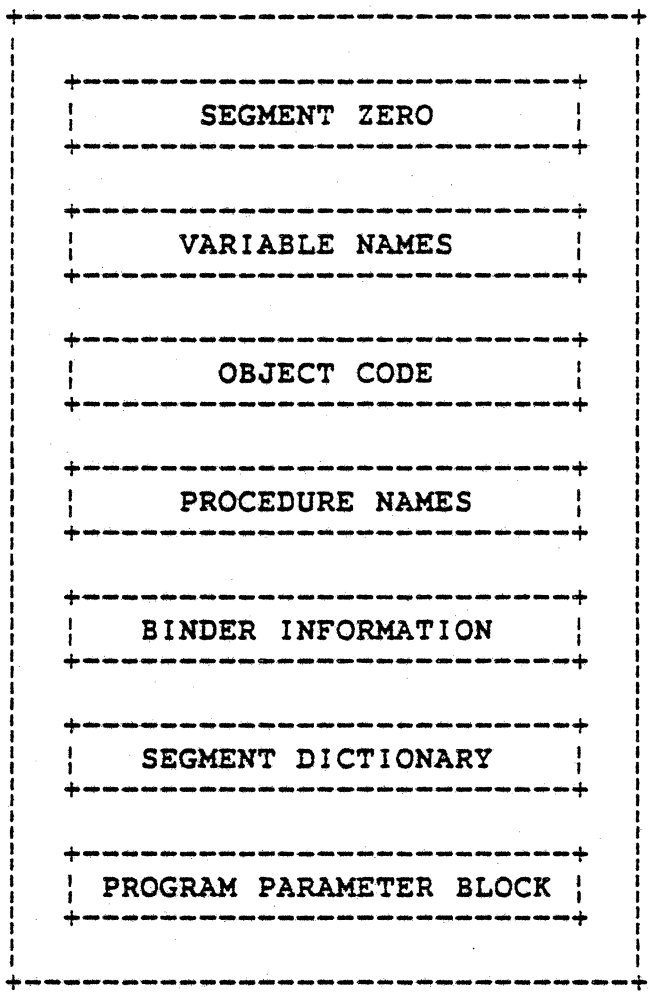


Figure 1-22. ALGOL Code File

1 INTRODUCTION

NEWP is an ALGOL-like language that has replaced ESPOL as the language in which the MCP is written. Although NEWP has been designed as a general-purpose language, the current NEWP compiler is specifically aimed at aiding the conversion of the MCP symbolic from ESPOL to NEWP. This document describes only the features of NEWP that appear in the current MCP symbolic. Subsequent releases of the NEWP compiler will be accompanied by documentation describing additional NEWP constructs.

The NEWP compiler will produce only non-executable (MCP) code files. Some features are described as "temporary". These features were implemented to facilitate the MCP conversion and will be deleted from the language as soon as it is practical to do so.

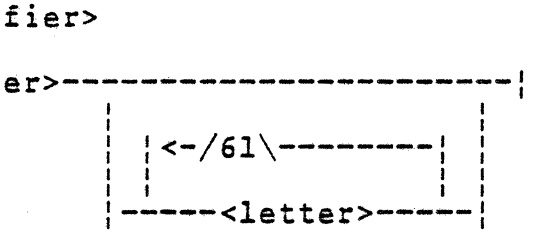
Aspects of NEWP which are not specified in this document are identical in operation to ALGOL as described in the B7000/B6000 ALGOL Language Reference Manual (Form No. 5001639). The NEWP implementation also includes the interface to libraries as it is implemented in ALGOL.

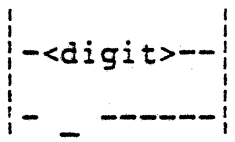
For details regarding those differences between ESPOL and NEWP that affected the MCP conversion, see "P2288 -- MCP Conversion to NEWP".

2 LANGUAGE COMPONENTS

All reserved words which have meaning only in limited contexts are type 3 reserved words. This type includes file attributes, task attributes, CLOSE options, PROGRAMDUMP options, carriage control parameters in I/O statements, and fault names. When context indicates that a type 3 reserved word is appropriate, the compiler first looks for a reserved word. If one is not found, defines are expanded and the compiler looks again for a reserved word.

The definition of identifiers has been changed to include underscore ("_") characters. The syntax diagram for an <identifier> is now the following:





Underscores are treated as significant characters in an identifier. INTERACTIVEXREF has been changed to allow it to reference these new identifiers.

3 PROGRAM STRUCTURE

In NEWP, lex levels are determined by procedure nesting, not block nesting as in ALGOL. Thus, a procedure which is not a block changes the lex level, while a block which is not a procedure does not.

Blocks which are not procedures are not entered with an ENTR operator in NEWP. Items which are declared in the block are simply added to the stack at the current lex level. Since this implementation requires less execution-time overhead, NEWP blocks are called "cheap" blocks.

Because cheap blocks are not entered via an ENTR operator, it is illegal to leave a block by performing an EXIT or RETURN. Cheap blocks do not change the addressing environment; thus, the lex level does not change and there is no MSCW for the block.

Segmentation is based upon lex levels. By default, a new segment is assigned to each procedure declared at lex level 15 or below (whether or not local variables are declared). The lex level at which segmentation occurs can be changed by using the SEGMENTLEVEL block direction. In addition, automatic segmentation can be overridden on a block-by-block basis by use of the SEGMENT block direction.

In NEWP, by default, procedures declared at lex levels less than or equal to the specified SEGMENTLEVEL are in separate segments, regardless of whether or not local variables are declared, and blocks are not in separate segments. Default segmentation can be overridden by using block directions, as described in the NEWP D-Note.

The ESPOL constructs BEGINSEGMENT, ENDSEGMENT, RESIDENT, CONTROL, and SAVE (for procedures) do not exist in NEWP. Instead, the SEGMENT and CONTROLSTATE block directions can be used to perform the same functions (e.g. an outer block "SAVE" procedure is specified in NEWP as [SEGMENT=5, CONTROLSTATE]).

Note: The problem that exists in ALGOL where variables declared at a low lex level with a high offset may become

inaccessible at higher lex levels has been alleviated in NEWP.

*See also
8.2 Block Directions

4 DECLARATIONS

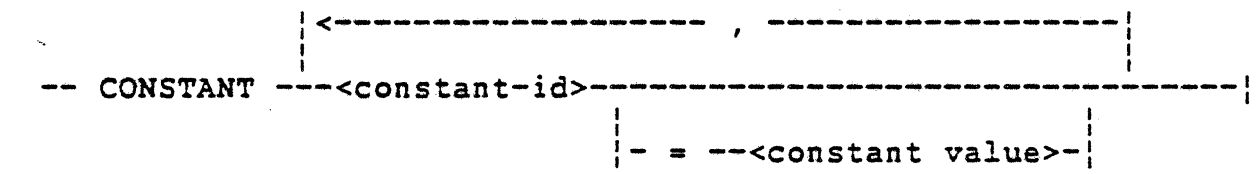
In addition to the differences and additions described in the following subsections, NEWP declarations differ from ALGOL declarations in the following aspect:

- DEFINE invocations are expanded within FORMAT declarations and in-line formats when an item is not recognized as a format specifier.

- DEFINE expansion will not occur when processing the individual identifiers in the <value part> or <specification part> of a procedure heading. This differs from ALGOL, which expands defines in these places.

4.1 CONSTANT DECLARATION

<constant declaration>



<constant value>

--<constant arithmetic expression>

Example:

```
CONSTANT
  TASKMSCW,              % AUTOMATICALLY ASSIGNED 0
  TASKPARAMS,            % 1
  CODEHEADERINDEX = 0,   % STARTS AGAIN AT 0
  RUNNINGCOUNT,          % 1
  CODELINKS,             % 2
  MARKER = CODELINKS,    % 2 ALSO
  COMPILERINFO,          % 3
  TOFFSET = MARKER+5,    % 7
```

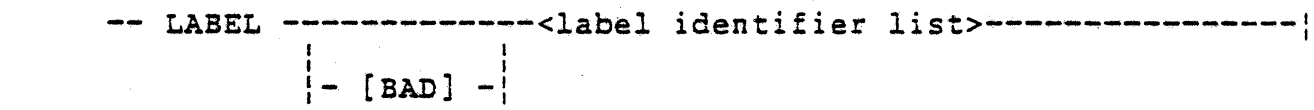
```
NEXTWORD;              % 8
```

The CONSTANT declaration can be used to declare arithmetic constants. It is particularly useful for declaring a series of integer constants where each value is to be one greater than the previous value.

If a <constant-id> appears with a <constant value>, then the identifier is associated with the specified value. If no <constant value> is given, the compiler assigns the identifier the value of the previous identifier plus one (or zero, if it is the first identifier in the list).

The <constant value> is evaluated in the context of the CONSTANT declaration (unlike defines, for which the text is expanded in the context of the invocation).

4.2 LABEL DECLARATION

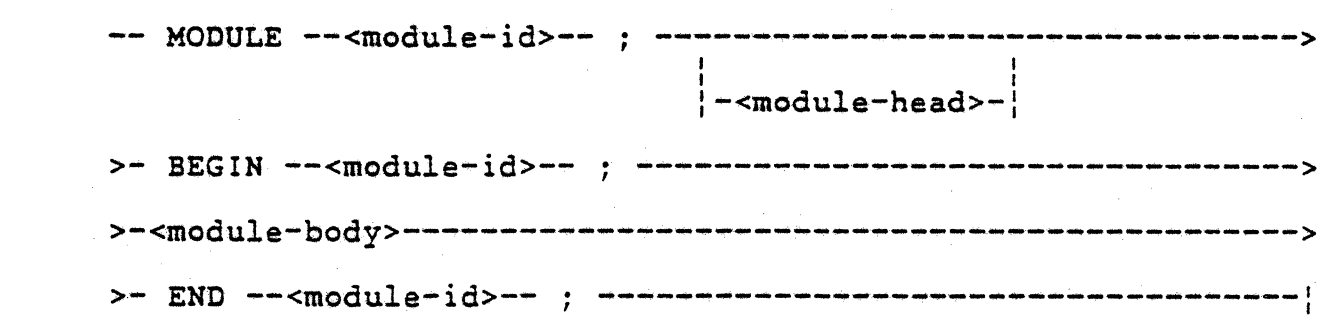


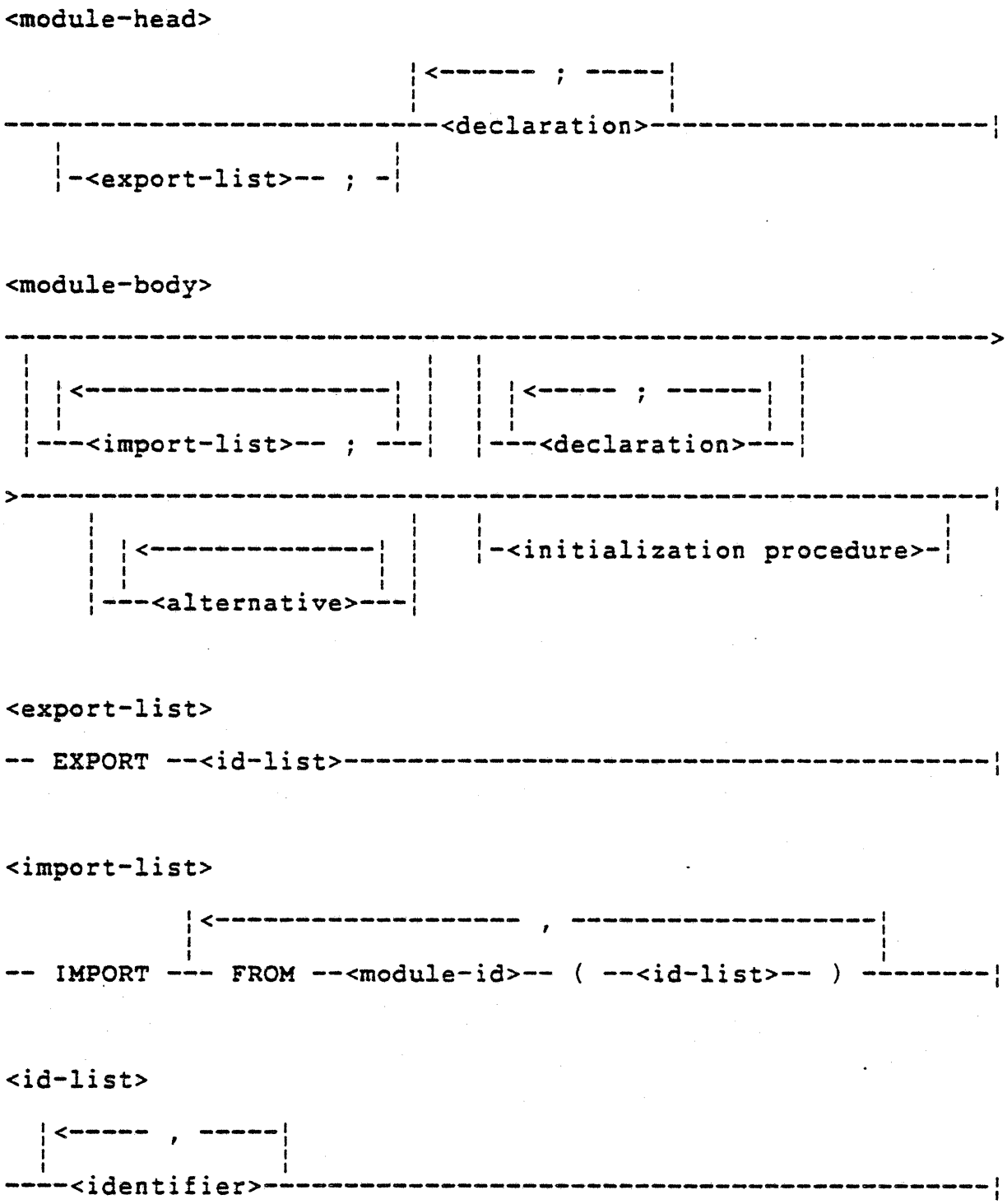
Example:

```
LABEL [BAD] ENDITALL, ERROREXIT;
```

In NEWP, a "bad go to" is a GO TO statement that branches out of the segment or procedure in which the GO TO statement appears. A label which is the object of a "bad go to" must be declared in a label declaration which includes the "[BAD]" syntax.

4.3 MODULE DECLARATION





The MODULE declaration allows logically-related declarations to be grouped together. Items declared within a module are "protected" in the sense that they are not visible to other modules unless specified in an EXPORT list; even if exported, items which are IMPORTed into another module may not be changed by that module.

EXPORTed identifiers must be declared in the <module-head>. Procedures to be EXPORTed are declared in the <module-head> as they would be declared anywhere else, except that the <procedure body> must be EXTERNAL, NULL, or FORWARD. If the <procedure body> is FORWARD, the procedure must be fully declared in the <module-body>.

A module that requires access to an EXPORTed identifier must specify that identifier in an IMPORT list. Access to IMPORTed variables is always read-only.

"MODULE <module-id>", "BEGIN <module-id>", and "END <module-id>" must be the first tokens on the card images on which they appear.

TEMPORARY -- As a temporary convenience, exported identifiers are implicitly imported into all modules; thus, IMPORT lists are temporarily optional.

TEMPORARY -- Temporarily, modules that import identifiers are allowed both read and write access to those identifiers.

Example of Modules:

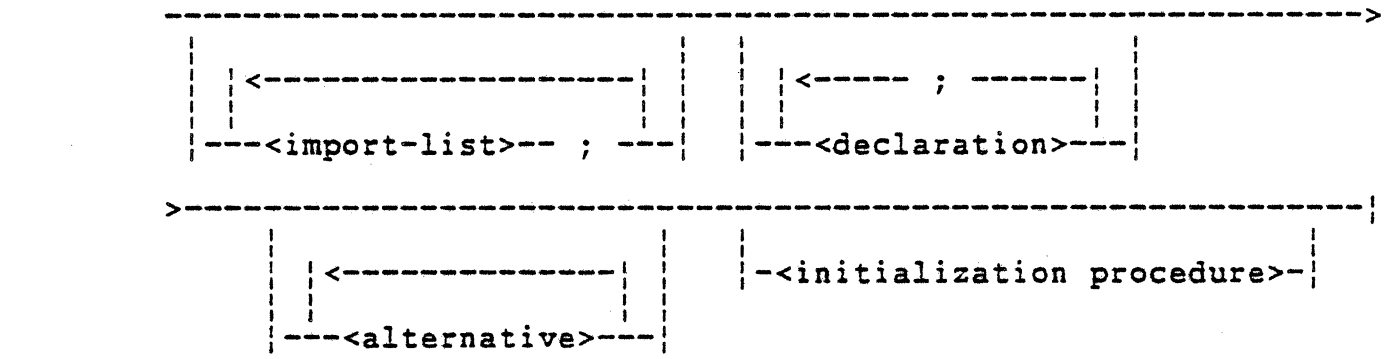
```
BEGIN
...
MODULE X;
  EXPORT RX, PX;
  REAL RX;
  PROCEDURE PX(B);
    VALUE B; REAL B;
    FORWARD;
BEGIN X;
  IMPORT FROM Y (RY, PY);
  PROCEDURE PX(B);
    VALUE B; REAL B;
    BEGIN
    RX := RY;
    PY;
    END;
END X;
...
MODULE Y;
  EXPORT RY, PY;
  REAL RY;
  PROCEDURE PY;
    FORWARD;
BEGIN Y;
  REAL LOCALR;
  IMPORT FROM X(RX);
  PROCEDURE PY;
    BEGIN
    LOCALR := RX;
    END;
```

```
    END Y;
    ...
    END.
```

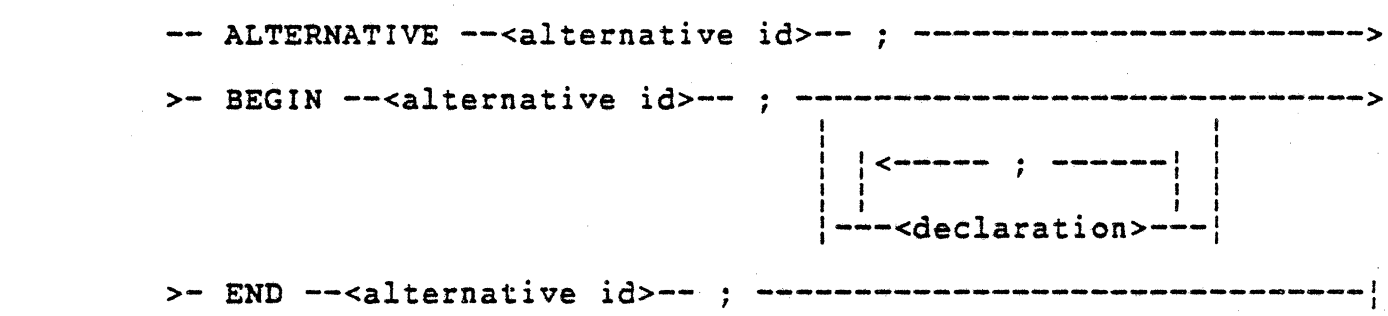
4.4 ALTERNATIVES AND INITIALIZATION PROCEDURES

Modules may now contain <initialization procedure>s and <alternative>s. Following is the new syntax for <module body>.

<module body>



<alternative>



If a module contains any alternatives, then an initialization procedure must appear last in the <module body>.

If an initialization procedure occurs in a module, all the items declared in the module, except the initialization procedure, are unavailable until the initialization procedure

is entered. At that time the items declared in the module but outside any alternatives are initialized. Items declared in an alternative are not available until and unless a select statement for that alternative is executed.

"INITIALIZATION" is now a reserved word; it cannot be used as a identifier.

Items declared in a module but outside any alternatives are available inside all the alternatives in that module. Therefore, if a procedure is declared forward outside the alternatives, its actual declaration must occur either outside the alternatives or inside each alternative.

Alternatives XREF in the same manner that modules XREF.

If a procedure has its forward in a module but outside any alternatives and its actual body in each alternative, each "alternative" procedure body is XREFed as an alias of each of the other "alternative" procedure bodies.

Alternatives may not contain external procedures.

Initialization Procedure

The differences between initialization procedures and other procedures are:

1. An initialization procedure may contain select statements.

2. An initialization procedure may only occur as the last declaration in a <module body>.

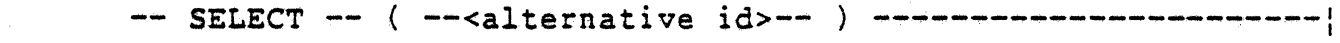
3. All items declared in the module, but outside any alternative modules that might exist, will be initialized when the initialization procedure is entered.

4. An initialization procedure may only be executed once. A run-time fault will occur if an attempt is made to execute it a second time.

5. The SEGMENT block direction may not be specified in the block directions for an initialization procedure.

Select Statement

<select statement>



The select statement initializes the module to include the the items declared inside the specified <alternative id>.

A select statement may only occur in an initialization procedure.

Only one select statement may be executed in an initialization procedure. A run-time fault will occur if a second select statement is executed.

Example:

```
BEGIN
MODULE PHYSICALIO;

EXPORT PHYSICALIOINIT,
       DOCHARIO,
       T;

BOOLEAN INITIALIZATION PROCEDURE PHYSICALIOINIT(WHICHONE);
VALUE WHICHONE; BOOLEAN WHICHONE;
   FORWARD;

PROCEDURE DOCHARIO;
   FORWARD;

INTEGER T;

BEGIN PHYSICALIO;
   REAL R;

   ALTERNATIVE HDPPHYSICALIO;
   BEGIN HDPPHYSICALIO;
      INTEGER I;

      PROCEDURE DOCHARIO;
         BEGIN
         END DOCHARIO;

   END HDPPHYSICALIO;

   ALTERNATIVE MPXPHYSICALIO;
   BEGIN MPXPHYSICALIO;

      PROCEDURE DOCHARIO;
         BEGIN
         END DOCHARIO;

      REAL PROCEDURE IOFINISH68;
         BEGIN
         END IOFINISH68;

   END MPXPHYSICALIO;
```

```
    BOOLEAN INITIALIZATION PROCEDURE PHYSICALIOINIT(WHICHONE);
    VALUE WHICHONE; BOOLEAN WHICHONE;
    BEGIN
       IF WHICHONE THEN
          SELECT(HDPPHYSICALIO)
       ELSE
          SELECT(MPXPHYSICALIO);
       PHYSICALIOINIT:=WHICHONE;

    END PHYSICALIOINIT;

END PHYSICALIO;

MODULE INITIALIZER;
BEGIN INITIALIZER;
    IMPORT FROM PHYSICALIO(PHYSICALIOINIT);
    BOOLEAN WHICHONE;

    PROCEDURE GETITGOING;
    BEGIN
       PHYSICALIOINIT(WHICHONE);
    END GETITGOING;

END INITIALIZER;

END.
```

4.5 ON DECLARATION

```
<on declaration>

-- ON --<fault list>-- , --<statement>-----------------------¦
```

Example:

```
ON ANYFAULT,
    BEGIN
    WRITE(TTYFILE,<"FAULT IN FIRSTPROC">);
    GO TO ERRLABEL;
    END;
```

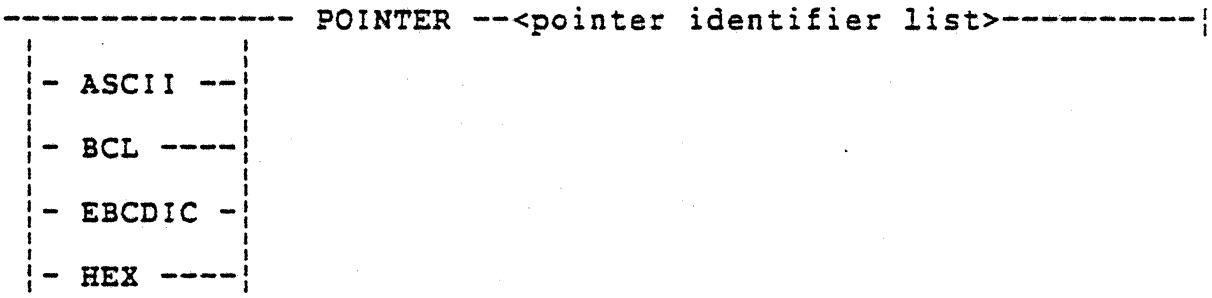
The ON declaration provides a mechanism for handling faults. When a fault named in the fault list occurs, control is transferred to the fault-handling statement appearing in the ON declaration. In order to resume normal execution, the programmer must perform a GO TO (except in the case of EXPONENT UNDERFLOW).

If a GO TO is not performed by the programmer, the system searches down the program's execution stack for another enabled fault-handling declaration. If none is found, the program is DSed (except if the fault was EXPONENT UNDERFLOW, in which case the program will continue processing with the operand that caused the underflow set to zero).

The <fault name>s and <fault number>s in NEWP are identical to those in ALGOL, except for the NEWP faults MEMORYFAIL1 (23), PARITYFAIL1 and PRIVILEGEDINSTRUCTION (24), which are available when the compiler option B7000 is set. Also, if B7000 is set, the following faults are not included in ANYFAULT, although they may be specified as individual <fault name>s: LOOP, MEMORYPARITY, INVALIDADDRESS, SCANPARITY, INVALIDPROGRAMWORD, MEMORYFAIL1, PARITYFAIL1.

A new fault name, LIBLINKFAULT, is now recognized by the compiler. This fault occurs during an unsuccessful attempt at linking libraries. The fault number is 21. This fault is trapped by the ON ANYFAULT declaration also.

4.6 POINTER DECLARATION



Example:

```
EBCDIC POINTER PTRIN, PTROUT;
```

Pointers may be declared with a size specification (e.g. HEX). If the character size is not specified, it defaults to EBCDIC.

Syntax errors are given for pointer and string size mismatches. For example, if PTR is declared as an EBCDIC pointer, the following statement will cause a syntax error:

```
REPLACE PTR BY 4"FF00";   % SHOULD BE 48"FF00"
```

4.7 PROCEDURE DECLARATION

Procedures in NEWP are similar to procedures in ALGOL, except 1) every <procedure body> must be delimited by a BEGIN/END pair and 2) parameters may not be passed by name.

Parameters may be passed as call-by-value or call-by-reference (call-by-name parameters and "thunks" are not implemented in NEWP). The default is call-by-reference. To pass a parameter as call-by-value, the <value part> must appear in the procedure heading.

Actual parameters passed to call-by-reference formal parameters must generate address references. Constants and arithmetic expressions do not generate address references. However, conditional and case expressions are allowed if each branch generates an address reference. For parameters passed by reference, the types of the actual and formal parameters must agree (e.g. a variable of type REAL cannot be passed by reference to a formal parameter of type DOUBLE or INTEGER). Procedures of type POINTER may be declared in NEWP.

The syntax for specifying procedures as formal parameters differs between NEWP and ALGOL. NEWP does not support run-time parameter checking; therefore, all parameters of formal procedures must be specified. The following diagram describes the syntax for the <specification> of a procedure which is a formal parameter (refer to the ALGOL Language Reference Manual page 4-55):

```
--<procedure type>-- PROCEDURE --<identifier>----------------->

>-<formal parameter part>-- ; -- FORMAL --------------------|
```

PROCEDURES WHICH HAVE ARRAYS DECLARED LOCAL TO THEM WILL GO THRU BLOCKEXIT. THE CALL ON BLOCKEXIT IS LAST THING IN THE PROCEDURE. TO AVOID BLOCKEXIT THE PROGRAMMER MAY FORGET HIS OWN ARRAYS AND DO AN EXIT AT THE END OF THE PROCEDURE. THE EXIT MUST BE THE LAST UNCONDITIONAL STATEMENT IN A PROCEDURE OR A CALL ON BLOCKEXIT WILL BE GENERATED. IF THE PROCEDURE IS TYPED THE PROGRAMMER SHOULD USE THE STATEMENT RETURN(VALUE).

PROCEDURES MAY BE TYPED AS IN ALGOL. IN ADDITION, PROCEDURES MAY BE DECLARED TYPE WORD, POINTER AND DESCRIPTOR IN NEWP.

Within the scope of a typed procedure, the value of the typed procedure may now be accessed via the following construct:

```
<procedure name>.VALUE
```

This construct may be used in expressions, assigned to, and address equated to.

Example:

```
REAL PROCEDURE PROC;
BEGIN [UNSAFE(MISC)]
   BOOLEAN B= PROC. VALUE;
   PROC.VALUE:=10;
   IF PROC.VALUE=20 THEN
        .
        .
        .
END PROC;
```

PARAMETERS MAY BE PASSED BY REFERENCE OR BY VALUE. THERE IS NO PASS BY NAME IN NEWP. THE FOLLOWING IS A LIST OPERAND TYPES AND HOW THEY ARE PASSED AS PARAMETERS.

OPERAND	VALUE	REFERENCE
REAL	OPERAND	SIRW
INTEGER	OPERAND	SIRW
BOOLEAN	OPERAND	SIRW
WORD	OPERAND	SIRW
EVENT	N/A	SIRW
POINTER	COPY DSCR	SIRW
ARRAY	N/A	COPY DSCR
FILE	N/A	SIRW
PROCEDURE	N/A	SIRW
DESCRIPTOR	COPY DSCR	SIRW

Arrays and descriptors as by-reference parameters are distinguished as follows:

1. Formal parameter as a descriptor:

   If an array row is the actual parameter, a reference to the one-word descriptor for the array row is passed (i.e., an SIRW to the descriptor). In this case, it is the descriptor in itself that is the object and may be directly modified.

2. Formal parameter as an array:

   The referenced passed is to the data segment of the array (i.e., a COPY descriptor is passed directly). The fact that a descriptor is used should be of no interest to the

programmer.

THE FOLLOWING IS A LIST OF DECLARATIONS WHICH ALLOCATE STORAGE. THIS LIST WILL DESCRIBE THE ATTRIBUTES OF EACH TYPE. BOTH SAFE AND UNSAFE TYPES ARE LISTED.

REAL - SAME AS ALGOL

SINGLE PRECISION OPERAND. 48 BITS OF THE WORD ARE USED. THE PROGRAMMER DOES NOT HAVE ACCESS TO THE TAG BITS. TAG BITS ARE 0.

INTEGER - SAME AS ALGOL

39 BITS OF THE WORD ARE USED. THE PROGRAMMER DOES NOT HAVE ACCESS TO THE TAG BITS. TAG BITS ARE 0.

BOOLEAN - SAME AS ALGOL

48 BITS OF THE WORD ARE USED FOR LOGICAL OPERATIONS SUCH AS AND, OR AND NOT. BIT 0 IS USED AS THE TRUE/FALSE BIT. THE OPERATORS "AND" AND "OR" ACT ON THE TAG BITS. THE PROGRAMMER DOES NOT HAVE ACCESS TO THE TAG BITS. TAG BITS ARE 0.

WORD

48 BITS OF THE WORD ARE USED. IN ADDITION, THE PROGRAMMER HAS ACCESS TO THE TAG BITS. NO TAG VALUE CAN BE ASSUMED EXCEPT AT DECLARATION. THE VALUE OF A WORD AT DECLARATION IS 0 (ALL BITS).

EVENT - SAME AS ALGOL

DOUBLE PRECISION WORD. THE PROGRAMMER DOES NOT HAVE DIRECT ACCESS TO THE WORD. ONLY STATEMENTS SUCH AS WAIT, PROCURE AND CAUSE MAY BE USED ON AN EVENT. THE TAG IS 2.

POINTER

AT DECLARATION A POINTER IS A WORD WITH 0 AND A TAG OF 6. WHEN THE POINTER IS ASSIGNED IT WILL BE A COPY DESCRIPTOR. POINTERS MAY BE DECLARED WITH A SIZE SPECIFICATION (ASCII, BCL, EBCDIC, HEX). THE DEFAULT SIZE IS EBCDIC.

ARRAY

ARRAYS CAN BE DECLARED WITH ANY VALID ALGOL TYPE. IN ADDITION, ARRAYS OF TYPE WORD AND DESCRIPTOR CAN BE DECLARED IN NEWP.

NEWP allows the indication that an array bound is unspecified. A bound pair of 0:-1 indicates an

unspecified bound. Except for this special case, the lower bound may not exceed the upper bound. If a dimension of an array is unspecified, it must either be the last dimension or all the dimensions must be unspecified.

When unspecified bounds are used, the MCP (ARRAYDEC) is called if only the last dimension of a multi-dimensional array is unspecified. For all other cases of unspecified bounds, an empty descriptor is built for the array.

ARRAYS CAN BE DECLARED WITH NO BOUNDS IF ADDRESS EQUATED. IN THIS CASE ONLY LOWER BOUND IS GIVEN WHICH MUST BE ZERO. ARRAYS MAY BE DECLARED SAVE. THAT IS, SAVE MEMORY WILL BE ALLOCATED FOR THE ARRAY WHEN IT IS TOUCHED.

FILE

FILES CAN BE DECLARED IN NEWP. AT DECLARATION A FILE WILL BE A NON PRESENT DATA DESCRIPTOR WHICH POINTS TO A DATA POOL. IF ADDRESS EQUATION IS USED WHEN THE FILE IS DECLARED AN ATTRIBUTE LIST MAY NOT BE USED. FILES MAY NOT BE DECLARED AT D0 BECAUSE THE SIRW TO LOCATE THE SEGMENT DICTIONARY STACK IS NOT BUILT AT D0.

PROCEDURES

PROCEDURES CAN BE DECLARED AS TYPED OR UNTYPED, WITH OR WITHOUT PARAMETERS. PARAMETERS CAN BE CALL BY VALUE OR CALL BY REFERENCE. PROCEDURES CAN BE DECLARED WITH OR WITHOUT A BODY. IN ADDITION, PROCEDURES CAN BE DECLARED FORWARD OR EXTERNAL.

DESCRIPTOR

VARIABLES DECLARED AS DESCRIPTOR ARE USED TO STORE UNINDEXED MOM OR COPY DESCRIPTORS. A DESCRIPTOR IS INITIALIZED TO 0 AND A TAG OF 6. THE PROGRAMMER CAN ACCESS THE TAG.

VALUE ARRAY

VALUE ARRAYS ARE DECLARED AS IN ALGOL. HOWEVER, THE ARRAY DESCRIPTOR IS ALLOCATED AT D0. IF THE OPTION MCP IS SET THE VALUE ARRAY IS ALLOCATED IN THE PROCEDURE IN WHICH IT IS DECLARED.

Inline procedures combine the efficiency of defines with the semantics of procedures. Each invocation of an inline procedure will result in an inline expansion of its code at the point of the invocation. Each allowable invocation of an inline procedure maintains the semantics of a procedure call.

The lineinfo for an expanded inline procedure will include both the sequence number(s) of the invoking code and the sequence number of the invoked code. These sequence numbers will be in order from local to global and separated by a slash ('/').

Syntax:

a). Declarations

The new keyword 'INLINE' will be a block directive allowed after the first 'BEGIN' of the procedure body of a procedure declaration.

Example:

```
REAL PROCEDURE PROC;
BEGIN [INLINE]
   :
   :
END PROC;
```

b). Modules

An inline procedure that is exported from a module must have its body occur in the module heading. A non-inline procedure may not have its body occur in the module heading.

Restrictions:

The following restrictions exist for inline procedures. Syntax errors will be issued if they are violated.

a). Recursion

Recursion of inline procedures is not allowed.

b). Parameters

An inline procedure may not be passed as a formal parameter to a non-inline procedure. However, both inline and non-inline procedures may be passed as formal parameters to an inline procedure.

c). IPC

IPC commands, (e.g. PROCESS, RUN, etc.), may not specify an inline procedure.

d). Libraries

An inline procedure may not be exported as a library.

e). Non-invocation references

Non-invocation references, (e.g. MAKEPCW, LEXOFFSET, etc.), to inline procedures are not allowed.

f). RETURN and EXIT

The RETURN and EXIT statements will not be allowed within the body of an inline procedure

g). Forwards

An inline procedure may not be declared forward.

h). Initialization procedures

An initialization procedure may not also be an inline procedure.

i). SORT statement

An inline procedure may not be used in a SORT statement.

j). Module visibility

As a temporary restriction, the only imported items an exported inline procedure may use are those exported from modules declared prior to the module containing the exported inline procedure.

4.8 SEGMENT IDENTIFIERS

A new type of identifer, the <segment identifier>, has been added It is used to refer to a code segment. <Segment identifier>s are declared in <segment declaration>s.

<segment declaration>

--- SEGMENT ---<identifier list>----------

An implicitly declared segment identifier, OUTERBLOCK, exists. It refers to the code segment that contains the outerblock code.

SEGMENT is not a general type. It can only be used as described below.

1. <segment identifier>s are allowed in block directions of the form SEGMENT = <segment identifier>.

2. Any variable that can normally be address equated, can be address equated to a <segment id>. Also, <segment id>s can be address equated but only to absolute addresses with lex level = 0 and displacement less than the fixed D0 fence.

   Examples:

```
SEGMENT SEG1 = (0,1),      % legal
        SEGA = (0,1000),   % syntax error if 1000 is
                           % above the fixed D0 fence
        SEGB = (1,2),      % syntax error because it's
                           % not at lex level 0
        SEGC = REALID;     % syntax error because it's
                           % relative, not absolute
                           % address equation
WORD    WSEG1 = SEG1;      % legal
```

3. <segment identifier>s may occur in export lists and import lists.

4. <segment identifier>s are allowed as the parameter to LEXOFFSET and as the <location designator> in <type> AT <location designator>.

Examples:

```
REALID:=LEXOFFSET( SEGID );
WORDID:=WORD AT SEGID;
```

A syntax error will be given if a <segment id> other than OUTERBLOCK is referenced with LEXOFFSET or AT but is never used in a SEGMENT block direction. This will prevent accessing a <segment id> when there is no code segment associated with it.

5 STATEMENTS

The following list describes the differences between NEWP statements and ALGOL statements. Page numbers refer to applicable documentation in the ALGOL Language Reference Manual.

- A <partial word part> may appear on the left-hand side of

an update replacement (page 5-4), as shown in the following example: X.[5:3] :=* + 1.

- The function of the DEALLOCATE statement in ALGOL (page 5-31) is performed by the RESIZE statement in NEWP (page 5-91), as follows: RESIZE(<array row>,DEALLOCATE).

- When an <arithmetic expression> appears as the <source part> in a REPLACE statement and no FOR clause is specified, exactly 48 or 96 bits of data (depending on whether the <arithmetic expression> is single-precision or double-precision) are transferred as characters of the size determined by the <destination part> (page 5-78). This implementation differs from ALGOL, where either 6 or 8 characters would be transferred, depending on the value of the BCL dollarcard option.

- The SET and RESET statements for events have been renamed SETEVENT and RESETEVENT (pages 5-98 and 5-90).

- The function performed by the REWIND statement in ALGOL (page 5-92) is performed by the CLOSE statement in NEWP (page 5-25), as follows: CLOSE(<file designator>,RETAIN).

- Branching into THRU loops (page 5-109) is not permitted. Branching within a THRU loop is allowed, provided the label is declared within the loop.

- The body of a FOR statement is now considered to be a new environment, thus preventing GOTO statements from branching from the outside to the inside of the body of a FOR statement.

- The <time> specified in a WAIT statement (page 5-114) need not appear within its own set of parentheses and need not appear first in the <wait parameter list>. If a <time> appears in the parameter list, it will be evaluated before any event parameters.

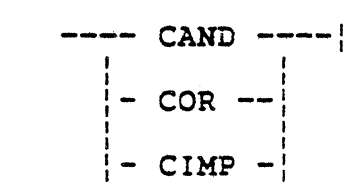
- The function of the WHEN statement (page 5-117) in ALGOL is performed by the WAIT statement (page 5-114) in NEWP, as follows: WAIT(<arithmetic expression>).

The NEWP language now has binary infix <conditional-operator>s which are used to combine <boolean-primary>s in boolean expressions in the same way that <logical-operator>s are used.

The <conditional-operator>s are similar to the <logical-operator>s except that the right-hand operand is not evaluated if the value of the left-hand operand is sufficient to determine the value of the operation.

Syntax:

<conditional operator>



Operands L	R	Operations L CAND R	L COR R	L CIMP R
TRUE	bool	bool	TRUE	bool
FALSE	bool	FALSE	bool	TRUE

The <conditional-operator> 'CAND' has the same precedence as the <logical-operator> 'AND', 'COR' the same precedence as 'OR', and 'CIMP' the same precedence as 'IMP'.

EXAMPLES:

```
B := R1 NEQ 0 CAND R2/R1 EQL R3;
B := R1 GEQ 0 AND R1 LSS SIZE(A) COR R2 NEQ A[R1];
B := R1 GTR 0 CIMP A[R1-1] = R1;
```

THE ASSIGNMENT STATEMENT IS THE SAME AS ALGOL. HOWEVER, THE NEWP PROGRAMMER SHOULD BE AWARE OF THE CODE GENERATED TO ACCESS A GIVEN DATA TYPE. THE FOLLOWING IS A TABLE OF DATA TYPES AND THE CODE GENERATED TO ACCESS THE DATA TYPE. BOTH SAFE AND UNSAFE TYPES ARE LISTED.

	OPERATION	
DATA TYPE	FETCH	STORE
REAL	VALC	NAMC STOD
WORD	NAMC LODT	NAMC OVRD A REG - IRW B REG - DATA
DESCRIPTOR	NAMC LOAD	NAMC OVRD
BOOLEAN	VALC	NAMC STOD
INTEGER	VALC	NGTR NAMC

		STOD
REAL ARRAY ONE DIMENSION	VALC(INDEX) VALC(DESCRIPTOR) OR VALC(INDEX) NAMC(DESCRIPTOR) NXLV	NAMC(DESCRIPTOR) VALC(INDEX) INDX STOD
REAL ARRAY TWO DIMENSION	VALC(INDEX 1) NAMC(DESCRIPTOR) NXLN VALC(INDEX 2) NXLV	NAMC(DESCRIPTOR) VALC(INDEX 1) NXLN VALC(INDEX 2) INDX STOD
WORD ARRAY ONE DIMENSION	VALC(INDEX) NAMC(DESCRIPTOR) INDX LODT	NAMC(DESCRIPTOR) VALC(INDEX) INDX OVRD A REG - DATA DESC B REG - DATA
WORD ARRAY TWO DIMENSION	VALC(INDEX 1) NAMC(DESCRIPTOR) NXLN VALC(INDEX 2) INDX LODT	NAMC(DESCRIPTOR) VALC(INDEX 1) NXLN VALC(INDEX 2) INDX OVRD

THE FOLLOWING IS AN ALGOL FEATURE. HOWEVER, MANY PEOPLE ARE NOT AWARE OF IT. ASSUME A REAL "R" HAS A CHARACTER IN THE LOW END OF IT. A PROGRAMMER WANTS TO REPLACE A POINTER "P" BY THIS CHARACTER. THE REPLACE HARDWARE WILL PICK UP CHARACTERS FROM THE HIGH END OF THE WORD. THUS, THE CHARACTER NEEDS TO BE SHIFTED. THIS IS DONE WITH A WRAP AROUND ISOLATE. THE FOLLOWING STATEMENT IS USED.

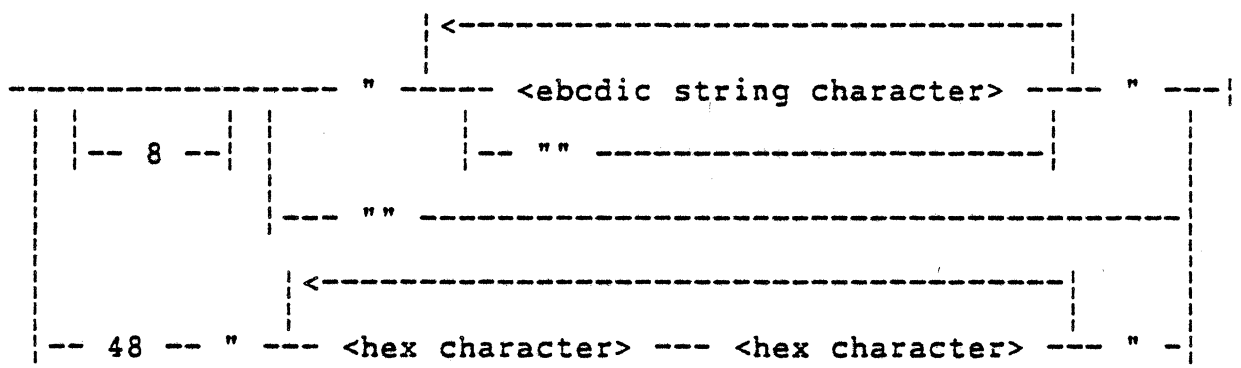
```
REPLACE P BY R.[7:48] FOR 1;
```

5.1 STRING AND NUMERIC CONSTANTS

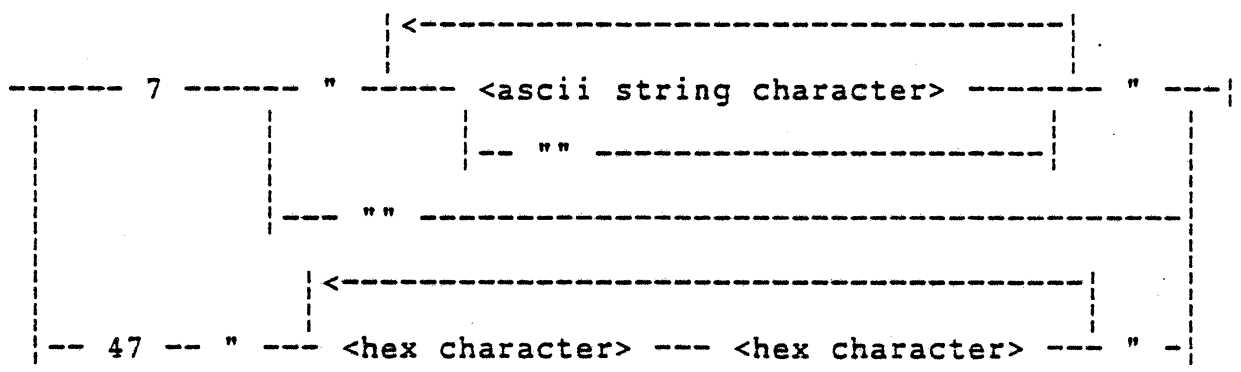
String constants and numeric constants have been added to NEWP. two language components are syntactically distinct, thus the ambiguity between string and numeric constants present in ALGOL has been eliminated.

STRING CONSTANTS

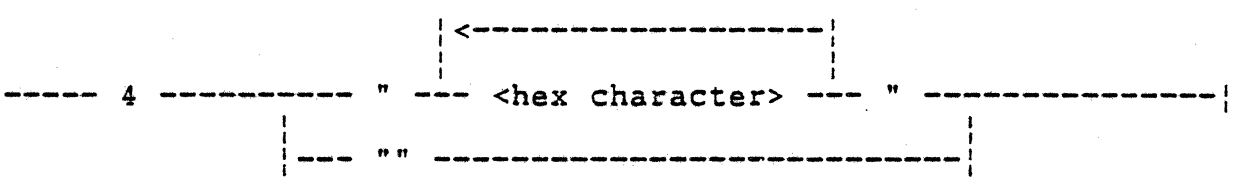
<ebcdic string>



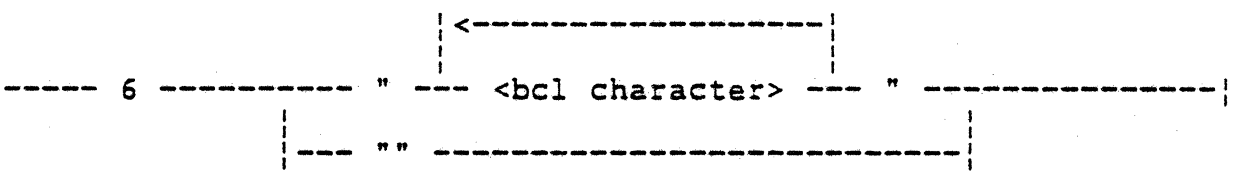
<ascii string>



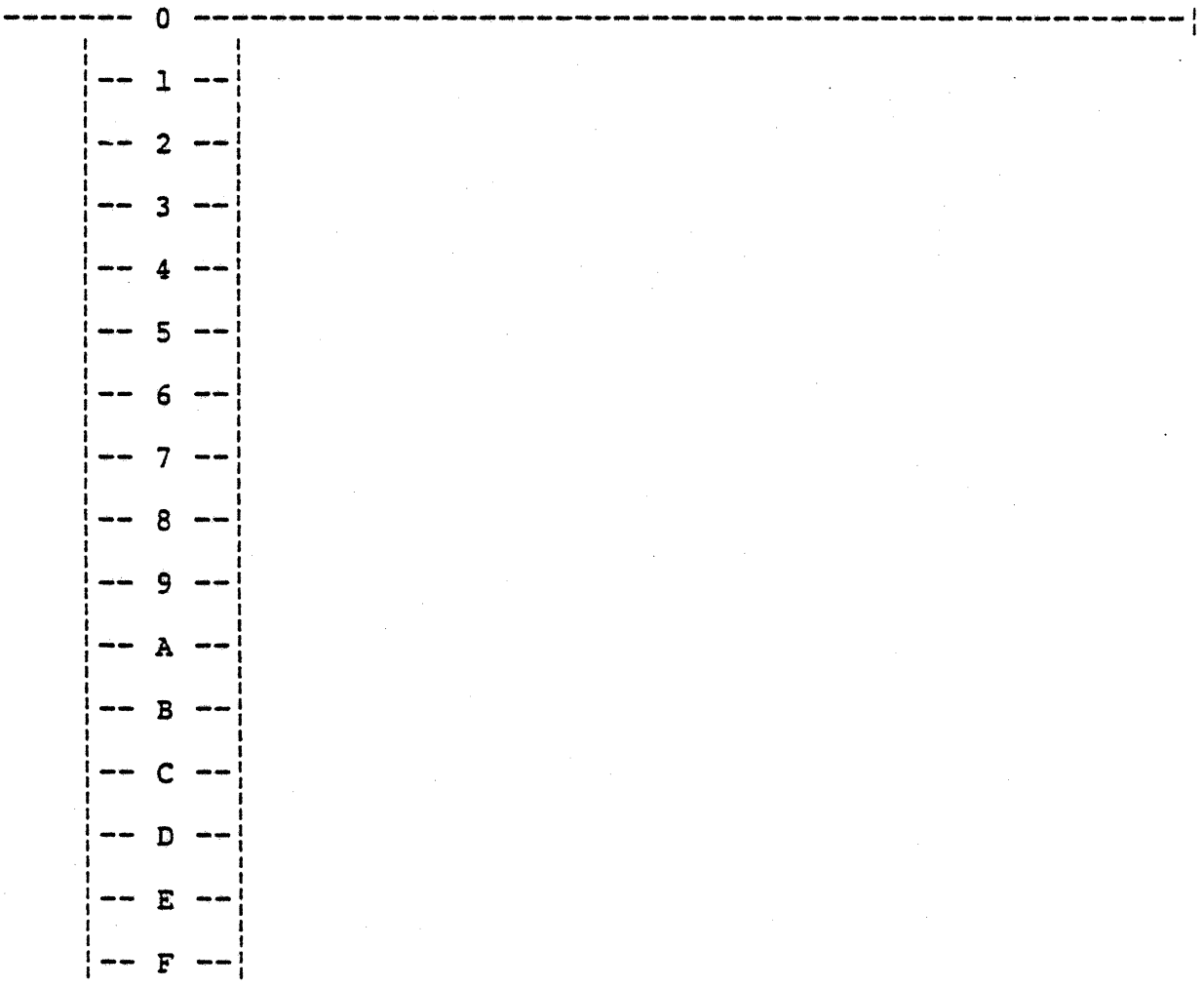
<hexadecimal string>



<bcl string>



<hex character>



```
<ebcdic string character>

----- any ebcdic character except a quote (") ----------------¦

<ascii string character>

----- any ascii character except a quote (")  ----------------¦

<bcl character>

--  any bcl character except a quote (")  --------------------¦
```

A string constant can be composed of EBCDIC(8-bit), ASCII(7-bit 8-bit format), or hexadecimal(4-bit) characters. A string constant is always left justified.

NOTE: The following are differences between NEWP and ALGOL constants.

1. There will not be any implicit concatenation in NEWP as in ALGOL. The concatenation symbol is required.

2. NEWP does not allow string constants to be considered as numeric constants without using the explicit type transfer function REAL.

3. NEWP uses the WFL rule for quoting a quote character not the ALGOL rule. To quote a quote character, two quote characters are required within a string constant.

4. An empty string is denoted by "" rather than the word EMPTY.

5. The "left justifying" prefixes (i.e. 80, 480, ...) are not implemented.

Examples

```
8"ABCD123";                    result = 'ABCD123'
"""WHY" & 48"6F" & """";       result = '"WHY?"'
"";                            result = an empty ebcdic string
7"";                           result = an empty ascii string
4"";                           result = an empty hex string
```

THE CONCATENATION OPERATION WITH STRING CONSTANTS

Two or more string constants may be concatenated together by using the <concatenation operator>. The concatenation of two

strings yields a new string whose length is the sum of the lengths of the two original strings. The value of the new string is formed by joining the second string immediately onto the end of the first string.

Only string constants of the same character type may be concated. If they are not of the same type, a syntax error occurs.

Examples

```
"STRING" & "CONSTANT"
"NUMBER" & 48"F1"
47"3138" & 7"ASCII CHARACTERS"
"" & "DOESNT DO MUCH"
```

THE REPLACE STATEMENT WITH <string constant> SOURCE

To make it possible to transfer data from a string to an array, <string constant> may be used as <source part> in a <source list> in a REPLACE statement.

Examples

```
REPLACE P BY "STRING CONSTANT";
REPLACE P BY "STRING NUMBER " & 48"F1";
```

TYPE TRANSFER FUNCTION

<REAL function>

```
-- REAL -- ( -- <string constant> -- ) ----------------------¦
```

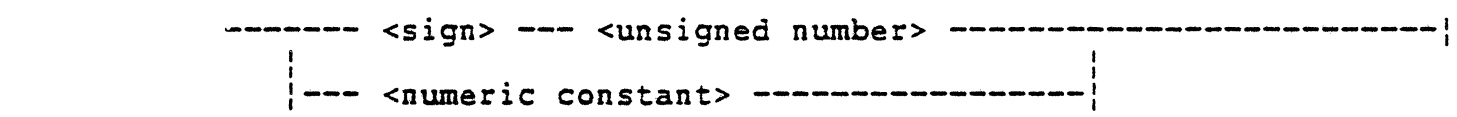
The REAL function returns as a real value the right justified bit image of the <string constant>. All bits in each character are used. The <string constant> may not exceed 48 bits in length.

Examples

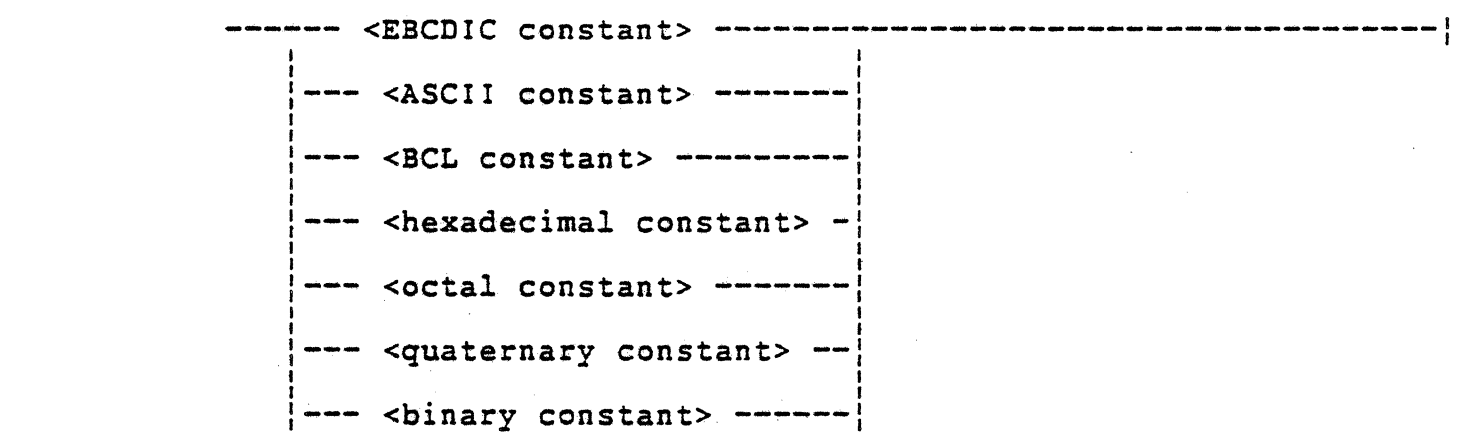
```
R := REAL("STRING");
R := REAL("TOO LONG");              compile-time error
```

NUMERIC CONSTANTS

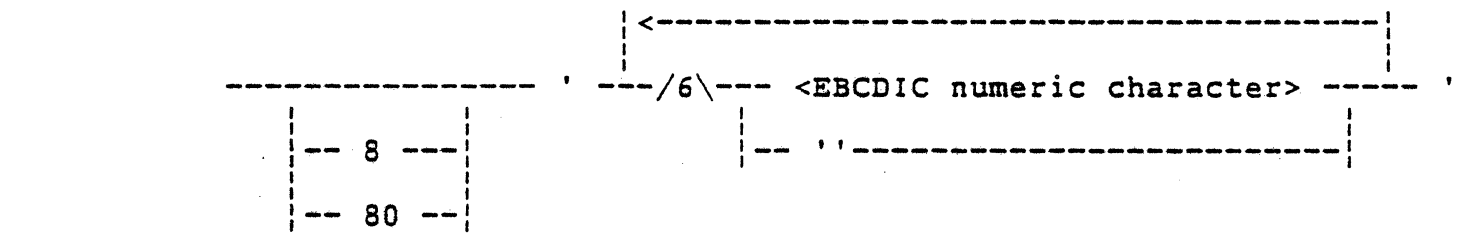
<number>



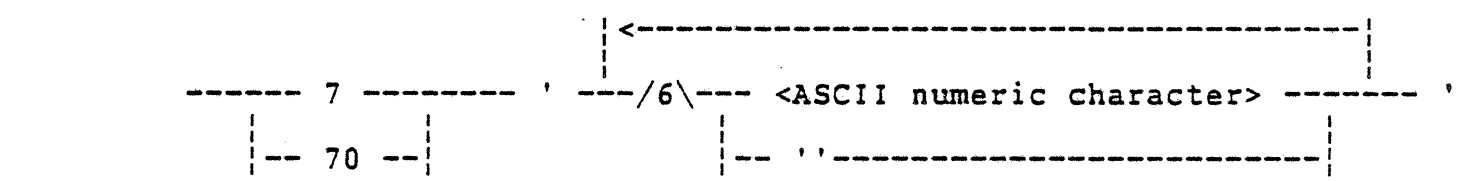
<numeric constant>



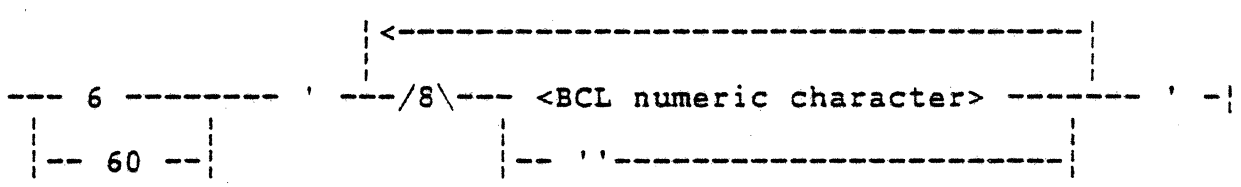
<EBCDIC constant>



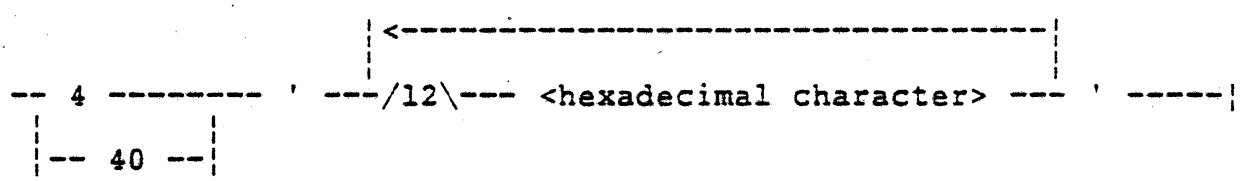
<ASCII constant>



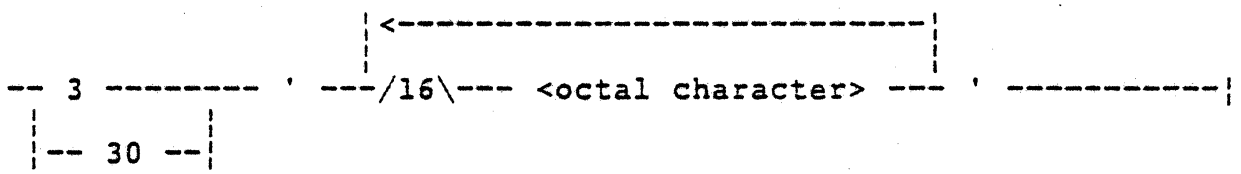
<BCL constant>



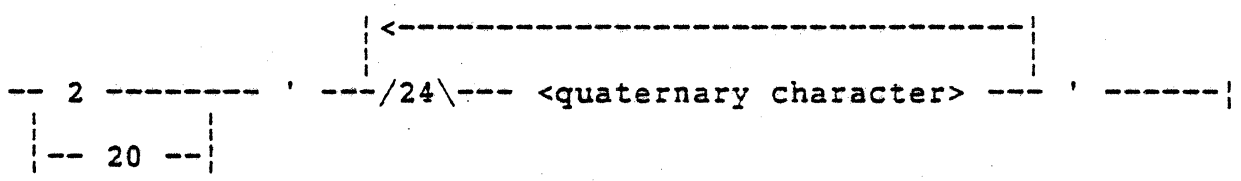
<hexadecimal constant>



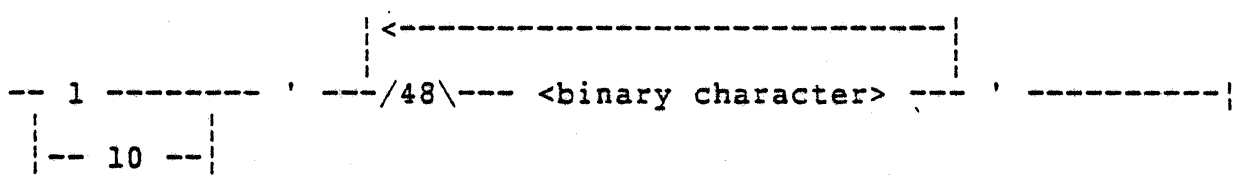
<octal constant>



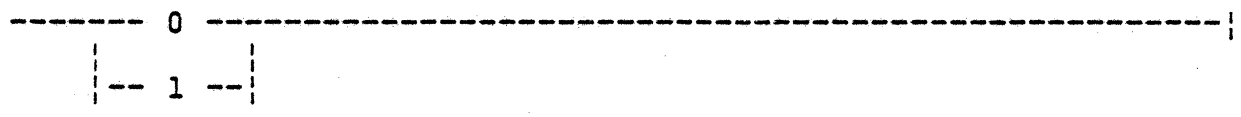
<quaternary constant>



<binary constant>



<binary character>



```
<quaternary character>

------- <binary character> ----------------------------------|
    |                         |
    |-- 2 --------------------|
    |                         |
    |-- 3 --------------------|

<octal character>

------- <quaternary character> ------------------------------|
    |                             |
    |-- 4 ------------------------|
    |                             |
    |-- 5 ------------------------|
    |                             |
    |-- 6 ------------------------|
    |                             |
    |-- 7 ------------------------|

<hexadecimal character>

------- <octal character> -----------------------------------|
    |                        |
    |-- 8 -------------------|
    |                        |
    |-- 9 -------------------|
    |                        |
    |-- A -------------------|
    |                        |
    |-- B -------------------|
    |                        |
    |-- C -------------------|
    |                        |
    |-- D -------------------|
    |                        |
    |-- E -------------------|
    |                        |
    |-- F -------------------|

<BCL numeric character>

-----  any BCL character except a single quote (')  ----------|

<ASCII numeric character>

-----  any ASCII character except a single quote (')  --------|

<EBCDIC numeric character>

-----  any EBCDIC character except a single quote (')  -------|
```

Numeric constants provide a way to specify a number as a bit mask (in 7 different bases or character sizes) in a general way which is not ambiguous with string constants and does not require any type transfer. This is accomplished by using the single quote (') character to specify a numeric constant.

To specify a BCL, EBCDIC, or ASCII numeric constant which contains a single quote, two adjacent single quotes are used.

As with ALGOL string constant semantics, left justification may be denoted by using a character code which is a multiple of 10.

Numeric constants may not be less than 1 bit, nor may they exceed 48 bits in size.

5.2 LIBRARIES

The following changes have been made to NEWP to allow the MCP to freeze as a library:

1. MCP is allowed as a value for the parameter to the FREEZE statement. This will only be allowed if the program being compiled has set the MCP dollar option.

2. FREEZE(MCP) will be allowed in a block with no library lists. It will export all entry points that occurred in library export lists at D0 prior to that point in the symbolic. Any subsequent D0 library export lists will get syntax errors. The FREEZE(TEMPORARY) and FREEZE(PERMANENT) cases of the FREEZE statement will continue to be allowed only in the same block as a library export list.

If a D0 library export list is changed during sepcomp then any procedures with FREEZE(MCP) statements must be touched so that the FREEZE statement will be recompiled.

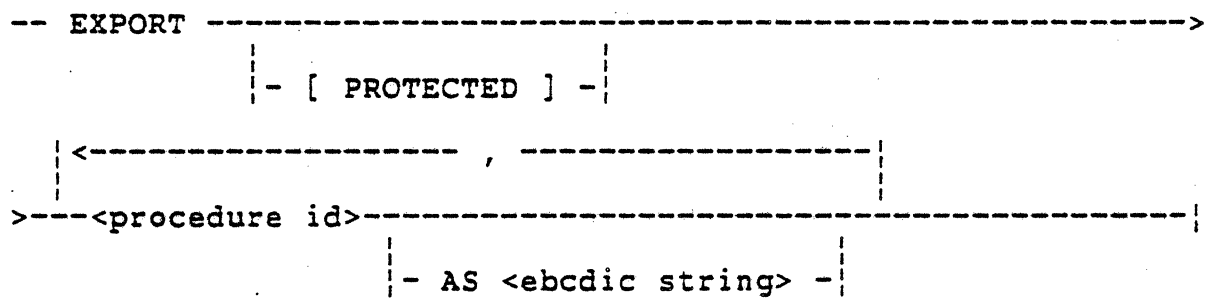
Note that an export list in a module head (i.e. between the 'MODULE <module-id>' and the 'BEGIN <module-id>') is a module export list. An export list in any other location is considered to be a library export list.

Library declarations may now specify the FUNCTIONNAME and LIBACCESS attributes. FUNCTIONNAME is a string-valued attribute used to specify the system function name that will be used to find the target code file for the library. LIBACCESS is a mnemonic-valued attribute: the value BYTITLE indicates that the TITLE attribute of the library is to be used to find the library's code file. The value BYFUNCTION indicates that the FUNCTIONNAME is to be looked up in the MCP library function table which is maintained by the ODT message

SL (see GENERAL note D3356) and the associated code file name will be used.

Library entry points declared in NEWP programs may now be exported with "protection". Such entry points may only be linked to by system libraries.

New Syntax:



This feature is intended to be replaced with more general protected constructs in a future release.

A new Boolean library attribute, SYSTEMLIB, has been added to NEWP When SYSTEMLIB is set, the associated library is to be initiated as a "system library"; therefore, it has access to protected MCP procedures. Use of this attribute requires the dollar option $MCP.

6 INTRINSICS

This section describes the intrinsics available in NEWP; page numbers refer to the ALGOL Language Reference Manual.

The following intrinsics are implemented in NEWP as they are in ALGOL (pages 6-19 to 6-30):

ABS, AVAILABLE
BOOLEAN
COMBINEPPBS, COMPILETIME
DAND, DEQV, DNOT, DOR, DOUBLE
FIRSTONE, FIRSTWORD
HAPPENED
INTEGER, INTEGERT
LISTLOOKUP
MASKSEARCH, MAX, MIN
NABS
OFFSET, ONES
POINTER, POTC, POTH, POTL
READLOCK, REAL(<ae>), REAL(<be>), REAL(<pe>,<ae>)

SECONDWORD, SINGLE, SIZE(<array designator>)
TIME

The function provided by the LINENUMBER intrinsic in ALGOL (page 6-24) is available as COMPILETIME(23) in NEWP.

The operation performed by the SCALERIGHTF intrinsic in ALGOL (page 6-26) is performed by the PACKDECIMAL intrinsic in NEWP.

Syntax:

```
-- PACKDECIMAL -- ( --<arithmetic expression 1>-- , -------->

>-<arithmetic expression 2>-- ) ----------------------------¦
```

The semantics of PACKDECIMAL are identical to the SCALERIGHTF function semantics in ALGOL.

The POINTER intrinsic in NEWP (page 6-30) allows the <character size> to be specified as 0, creating a word pointer which is considered as EBCDIC when pointer and string sizes are being matched.

The DAWDLE intrinsic in NEWP is an untyped intrinsic that takes an integer parameter. DAWDLE is used in the MCP to delay without accessing memory. The integer parameter specifies the number of Count Binary Ones (CBON) operators that are to be executed to effect the delay.

*See also
4.5 POINTER Declaration

7 UNSAFE MODE

Some constructs which the MCP requires in order to perform hardware-related functions are considered unsafe for general use. NEWP requires the MCP programmer to specify when and which unsafe constructs are to be used. This specification is made on a block-by-block basis through the use of the UNSAFE block direction. Programs that use the UNSAFE block direction are marked as non-executable.

*See also
8.2 Block Directions

7.1 DECLARATIONS

There are two additional data types available in UNSAFE mode, DESCRIPTOR and WORD.

DESCRIPTOR

> In UNSAFE(DESCRIPTOR) mode, DESCRIPTOR variables are allowed. The DESCRIPTOR data type is described in the following subsection.

WORD

> In UNSAFE(WORD) mode, WORD variables are allowed, with basically the same syntax and semantics as in ESPOL. Implicit type transfers ("coercions") between type WORD and other data types is more restricted in NEWP than in ESPOL. In particular, WORD/BOOLEAN coercion is disallowed in assignment operations.

In UNSAFE(MISC) mode, address equation as it exists in ESPOL is allowed in NEWP. Also, a <procedure body> can be specified as NULL, as in ESPOL, but only if the procedure has been address-equated. Array declarations may include the SAVE specification, as in ESPOL.

ITEMS MAY BE ADDRESS EQUATED IN NEWP. ITEMS MAY BE ADDRESS EQUATED TO EACH OTHER OR TO A GIVEN ADDRESS COUPLE. FOR EXAMPLE:

```
REAL R;
BOOLEAN BR = R;

ARRAY A[0:0];
DESCRIPTOR ARRAY DESCA = A [0];
ARRAY ATWO = A [0,0];
WORD WA = A;
DESCRIPTOR DA = A;

PROCEDURE P = (0,3);
BEGIN END;

WORD WP;
PROCEDURE P = WP;
NULL;

REAL PROCEDURE PQ(A,B);
REAL A,B;
```

```
BEGIN
   REAL MYVALUE = B+1;
   WORD MYMSCW = (1,0),
        MYRCW = MYMSCW + 1;
END;
```

7.1.1 DESCRIPTOR Data Type

Variables of type DESCRIPTOR are used to store unindexed mom or copy descriptors. Simple variables of type DESCRIPTOR are declared in a DESCRIPTOR declaration (similar to a REAL, INTEGER, or BOOLEAN declaration). Arrays, procedures, and formal parameters may also be specified as type DESCRIPTOR. Variables of type DESCRIPTOR are initialized to "uninitialized operand".

When values of type DESCRIPTOR are evaluated, copy bit action occurs if the target is a data descriptor.

The type transfer function DESCRIPTOR can be applied to both WORD variables and array rows. Implicit type transfers ("coercions") allow array references to be assigned from DESCRIPTOR values and DESCRIPTORs to be assigned from array references or array rows. The same coercions are applied between formal and actual parameters.

TEMPORARY -- Temporarily, WORDs and DESCRIPTORs are mutually coerced.

WHEN WORKING WITH DESCRIPTORS PROGRAMMERS MUST BE AWARE OF THE DIFFERENCES BETWEEN LOAD AND LODT OPERATIONS. THE LOAD OPERATION WILL CAUSE COPY DESCRIPTOR ACTION BUT LODT WILL NOT. IF D IS A DESCRIPTOR THEN THE STATEMENT "IF BOOLEAN(D.[46:1]) THEN ..." IS TRUE.

ARRAY IDENTIFIERS MAY NOT BE USED TO ACCESS DESCRIPTORS. IF A PROGRAMMER WANTS ACCESS TO THE DESCRIPTOR FOR AN ARRAY THE ARRAY SHOULD BE ADDRESS EQUATED TO A DESCRIPTOR OR THE TYPE TRANSFER INTRINSIC SHOULD BE USED.

7.2 STATEMENTS

The FORK statement is allowed in UNSAFE(FORK) mode, with similar syntax and semantics to ESPOL. A new required parameter, <box number>, must appear as the first parameter in the parameter list.

THE FORK STATEMENT IS USED TO START AN ASYNCHRONOUS PROCESS WITHIN THE MCP. THESE PROCESSES ARE CALLED MCP INDEPENDENT RUNNERS (IR). THE IR WILL BE A PROCEDURE IN THE MCP. THE PROCEDURE MAY HAVE PARAMETERS. SYNTAX FOR THE FORK STATEMENT

IS:

```
FORK PROCID [BOXNO,STACKSIZE,PRIORITY,NAME]

    BOXNO     - BOX THE IR IS TO RUN IN.
    STACKSIZE - STACKSIZE THE IR IS TO USE.
                IF BIT 47:1 IS SET THE IR WILL BE A CONTROL
                PROGRAM. IF BIT 46:1 IS SET THE IR IS A ONE
                ONLY INDEPENDENT RUNNER.
    PRIORITY  - PRIORITY THE IR IS TO USE. IF BIT 46:1 IS
                SET THE IR WILL BE VISIBLE ON THE ODT.
    NAME      - IF THIS PARAMETER IS PRESENT IT IS THE NAME THE
                IR IS TO USE. THE NAME PARAMETER IS A POINTER.
                FORMAT OF THE NAME IS LENGTH IN FIRST BYTE
                FOLLOWED BY THE TEXT (48"09""MCPIRNAME").
```

The OVERWRITE option on the REPLACE statement (as in ESPOL) is allowed in UNSAFE(MISC) mode.

SYNTAX FOR THIS OPTION IS:

```
    REPLACE DESTID BY SOURCEID FOR WORDCOUNT OVERWRITE
```

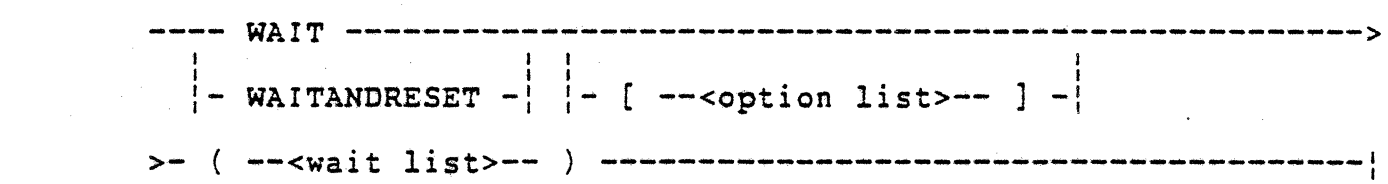
THIS STATEMENT WILL MOVE WORDCOUNT WORDS FROM SOURCEID (POINTER ID OR WORD ITEM) TO DESTID. THE OVERWRITE OPTION WILL ALLOW THE PROGRAMMER TO WRITE OVER ODD TAG (PROTECTED) WORDS.

In UNSAFE(REGISTERS) mode, the D-register override clause ("@ [<exp>]") is allowed following EXIT and RETURN (e.g. "RETURN @ [TASKADDR]").

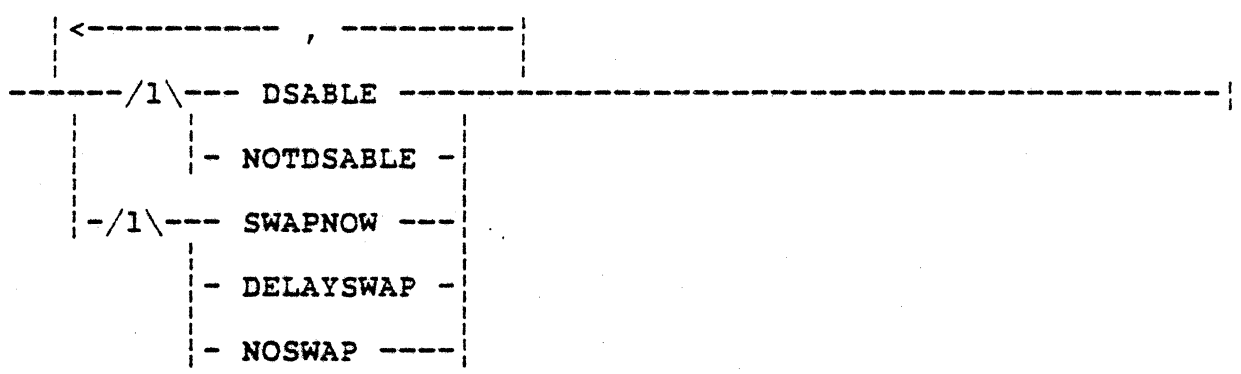
THIS SYNTAX WILL FORCE THE D REGISTER OF CURRENT LEX LEVEL TO POINT TO THE ADDRESS GIVEN. THUS, IT WILL ALLOW AN EXIT TO BE DONE BELOW THE NORMAL MARK STACK CONTROL WORD.

WAIT and WAITANDRESET statements may have option lists in UNSAFE(MISC) mode. This syntax is described in the following subsection.

7.2.1 WAIT Statement



<option list>



In UNSAFE(MISC) mode, the WAIT and WAITANDRESET statements may include an <option list> to specify whether or not the waiting process can be DSed or swapped out while waiting, as follows:

DSABLE

WAIT and WAITANDRESET with [DSABLE] perform the same functions that DSWAIT and DSWAITANDRESET do in ESPOL. The process will not wait if it is already DSed or will not continue to wait if it is externally DSed while waiting. The value returned by the WAIT function (if it occurs in an arithmetic expression) will be zero in either case.

NOTDSABLE

The process will wait even if it is or becomes DSed.

DELAYSWAP

The process will be swapped out if it waits longer than the maximum swap wait, defined in the MCP as MAXSWAPWAIT.

SWAPNOW

The process is swapped out as soon as the wait starts.

NOSWAP

The process must not be swapped out while waiting.

If an <option list> is not given, default values are assigned according to the type of process. System processes (D[0] and pseudo-D[0] relative code) are NOTDSABLE and NOSWAP. User processes default to DELAYSWAP and DSABLE (if not already

DSed; if the process has already been DSed, it will wait).

7.3 INTRINSICS

The following list describes the intrinsic identifiers recognized in UNSAFE mode. The UNSAFE category for each intrinsic is included in parentheses.

AT (REFERENCE)

The syntax "<type> AT <location designator>" allows the item at the location specified by the <location designator> to be referenced or assigned to as if it had been declared of the specified <type>. <Type> must be one of the following simple types: BOOLEAN, INTEGER, REAL, DOUBLE, POINTER, EVENT, DESCRIPTOR, FILE, TASK, WORD. <Location designator> can be a data item, a library identifier, a partially subscripted array (the descriptor is accessed), or a procedure identifier (the PCW for the procedure is accessed). If the AT syntax is used in an expression, the item is fetched from the location in the manner appropriate for a value of the target <type>.

For example, if D is a variable of type DESCRIPTOR, the syntax "WORD(D)" causes D to be fetched as a DESCRIPTOR, and copy bit action is performed. The syntax "WORD AT D", on the other hand, causes the item at D to be fetched as a WORD, and no copy bit action is performed.

Note that if the specified <location> is a formal parameter, "<type> AT <location>" references the actual parameter, not the local SIRW.

THIS STATEMENT WILL ALLOW TYPE TRANSFERS. THAT IS, THE PROGRAMMER CAN FETCH AND STORE ITEMS AS IF THEY ARE OF THE TYPE GIVEN. A FEW EXAMPLES OF THIS STATEMENT AND THE CODE GENERATED FOLLOW.

```
REAL R;  WORD W;  DESCRIPTOR D;

R:=REAL AT W;
   VALC(W)
   NAMC(R)
   STOD

D:=DESCRIPTOR AT W;
   NAMC(D)
   NAMC(W)
   LOAD
   EXCH
```

```
    OVRD

W:=WORD AT R;
    NAMC(W)
    NAMC(R)
    LODT
    EXCH
    OVRD
```

BUZZ (MISC)

The BUZZ intrinsic is implemented in NEWP as in ESPOL, with the additional restriction that BUZZ can be used only in CONTROLSTATE blocks.

THE BUZZ STATEMENT IS USED TO LOCK HARD LOCKS. IN GENERAL, THE STATEMENT BUZZ(LOCKID) IS THE FOLLOWING CODE:

```
WHILE READLOCK(1,LOCKID) DO
    SCALERIGHTF(ONES(1),2);
```

BUZZ47 (MISC)

BUZZ47 performs the same function that BUZZ does, except that bit 47 of the parameter is tested instead of bit 0. BUZZ47 is valid only when the B7000 compiler option is set.

DESCRIPTOR (DESCRIPTOR)

DESCRIPTOR is a type transfer function that can be applied to a WORD variable or an array row.

EVAL (MACHINEOPS)

TEMPORARY -- EVAL(<param>) in NEWP performs the same function that EVAL(NAME(<param>)) performs in ESPOL. EVAL is a temporary feature.

EXIT (MACHINEOPS)

EXIT is implemented in NEWP as it is in ESPOL.

FAILREGISTER (MACHINEOPS)

FAILREGISTER(<integer>) generates the Fetch Main Memory Fail Register (FMFR) operator for B7000 Series machines. The <integer> parameter is the memory module number. FAILREGISTER is valid only

if the B7000 compiler option is set.

FMMRREADLOCK (MACHINEOPS)

FMMRREADLOCK performs the same function that READLOCK does, except that the compiler emits the B7000 operator Fetch Main Memory Reference (FMMR) prior to emitting the Read With Lock (RDLK) operator. FMMRREADLOCK is valid only if the B7000 compiler option is set.

HEYOU (MACHINEOPS)

HEYOU is implemented in NEWP as it is in ESPOL.

THIS INTRINSIC GENERATES THE HEYU INSTRUCTION. THIS INSTRUCTION WILL INTERRUPT OTHER PROCESSORS. THIS INTRINSIC IS VALID ONLY ON A B6000 MACHINE.

IGNOREPARITY (MACHINEOPS)

IGNOREPARITY is an untyped procedure that generates the B7000 operator Ignore Parity (IGPR). IGNOREPARITY is valid only if the B7000 compiler option is set.

INTERRUPTCHANNEL (MACHINEOPS)

INTERRUPTCHANNEL(<real>) is an untyped procedure that generates the B7000 operator Interrupt Channel (INCN), where the <real> parameter is a mask indicating which channel is to be interrupted. INTERRUPTCHANNEL is valid only if the B7000 compiler option is set.

LEXOFFSET (MISC)

LEXOFFSET(<identifier>), where <identifier> represents a data item with an address couple, returns the MSCW-relative offset of that data item. For example, if X is declared at (1,9), LEXOFFSET(X) is 9.

MAKEPCW (MACHINEOPS)

TEMPORARY -- MAKEPCW is implemented in NEWP as it is in ESPOL. MAKEPCW is a temporary feature.

This note describes the use of and restrictions on the unsafe NEWP function MAKEPCW.

MAKEPCW accepts as a parameter either a procedure id or a label id. The result returned is a word value containing the hardware PCW pointing to the code for the procedure or label. This PCW will

properly have the NCSF field set to one for control state procedures and for labels declared in control state environments.

Restrictions on the use of procedure ids:

1. The procedure must not be declared EXTERNAL, NULL, LIBRARY, BY CALLING, or INLINE.

2. The procedure must not still be FORWARD at the time of the MAKEPCW invocation.

3. The MAKEPCW invocation cannot occur within the body of the procedure being passed as the parameter to MAKEPCW.

Restrictions on the use of label ids:

1. The declaration of the label must not be more global than the beginning of the code segment in which the label is used as a MAKEPCW parameter.

2. The PCW generated points to the actual label occurrence rather than to any hidden label generated for "bad GO TO" optimizations.

MEMORY (MEMORY)

MEMORY is implemented in NEWP as it is in ESPOL, except that M is not recognized as a synonym.

THE MEMORY ARRAY POINTS TO ALL OF MAIN MEMORY. THE DESCRIPTOR FOR THIS ARRAY IS AT (0,4). EXAMPLES OF THE USE OF THIS ARRAY FOLLOW.

```
MEMORY[5]:=W;
    NAMC(W)
    LODT
    NAMC(0,4)
    LT8 5
    INDX
    OVRD

W:=MEMORY[5];
    LT8 5
    LODT
    NAMC(W)
    OVRD

D:=DESCRIPTOR AT MEMORY[5];
    NAMC(D)
    LT8 5
    NAMC(0,4)
    INDX
```

```
LOAD
EXCH
OVRD
```

MOVESTACK (MACHINEOPS)

MOVESTACK is implemented in NEWP as it is in ESPOL.

THIS INTRINSIC (MOVESTACK(SNR)) WILL GENERATE THE MVST OPERATOR. THAT IS, IT WILL MOVE ADDRESS REGISTERS TO THE STACK SNR AND PLACE A TOSCW IN THE BASE OF THE OLD STACK.

PAUSE (MACHINEOPS)

PAUSE generates the IDLE operator. If two parameters are specified, the two values are loaded into the A and B registers before the IDLE operator is executed and are deleted afterwards.

POINTER(<word exp>) (WORD)

POINTER(<word exp>) performs a type transfer between type WORD and type POINTER.

REFERENCE TO (REFERENCE and WORD)

The function represented by the syntax "REFERENCE TO <primary>" returns a WORD which is equivalent to the value that would be generated to access <primary> if it were a call-by-reference parameter (e.g. an SIRW for simple data type, a data descriptor for array rows, an indexed data descriptor for subscripted variables). <Primary> can be of any data type that NEWP allows to be passed as a call-by-reference parameter.

EXAMPLES OF THE USE OF THIS INTRINSIC FOLLOW.

```
W:=REFERENCE TO R;
    NAMC(R)
    STFF
    NAMC(W)
    OVRD

W:=REFERENCE TO A[5]
    LT8 5
    NAMC(A)
    INDX
    NAMC(W)
    OVRD
```

```
W:=REFERENCE TO EVENT AT A[5];
   LT8 5
   NAMC(A)
   INDX
   XTND
   NAMC(W)
   OVRD
```

REGISTERS (REGISTERS)

REGISTERS is implemented in NEWP as it is in ESPOL.

THIS INTRINSIC CAN BE USED ACCESS PROCESSOR REGISTERS. THE INDEX PASSED IS THE REGISTER NUMBER. THIS NUMBER MUST BE A INTEGER CONSTANT. THE INDEX VALUES CAN BE FOUND ON P 3-99 OF B7700 INFORMATION PROCESSING SYSTEMS REFERENCE MANUAL (FORM # 1060233) OR P 8-9 OF B6800 INFORMATION PROCESSING SYSTEMS REFERENCE MANUAL (FORM # 5001290).

DLL (REGISTERS)

A new intrinsic, DLL, has been added which references the register D[LL]. DLL is available under UNSAFE(REGISTERS).

RETURN (MACHINEOPS)

RETURN is implemented in NEWP as it is in ESPOL.

THIS INTRINSIC GENERATES THE RETN INSTRUCTION. FOR EXAMPLE RETURN(5) WOULD GENERATE THE CODE LT8 5 RETN.

SCANIN (MACHINEOPS)

SCANIN is implemented in NEWP as it is in ESPOL.

THIS INTRINSIC GENERATES THE SCNI INSTRUCTION. THE PARAMETER PASSED TO THE INTRINSIC IS THE SCAN IN FUNCTION CODE.

SCANOUT (MACHINEOPS)

SCANOUT is implemented in NEWP as it is in ESPOL, except that it is not valid when the B7000 compiler option is set.

THIS INTRINSIC GENERATES THE SCNO INSTRUCTION. THIS INTRINSIC HAS TWO PARAMETERS. THE FIRST IS VALUE TO BE SCANED OUT (EX. AREA DESCRIPTOR, TIME OF DAY). THE SECOND IS THE SCAN OUT FUNCTION CODE.

SETINHIBIT (MACHINEOPS)

SETINHIBIT(<real>,<integer>) generates the B7000 operator Set Memory Inhibits (SINH), where the low-order eight bits of the <real> parameter contain the inhibit mask and the low-order four bits of the <integer> parameter represent the memory module. SETINHIBIT is valid only when the B7000 compiler option is set.

SETLIMITS (MACHINEOPS)

The number of parameters to the SETLIMITS intrinsic has been changed from four to two. SETLIMITS (<REAL>, <INTEGER>) generates the B7000 operator SLMT when the B7000 compiler option is set. With MOD3 MCM bits [7:8], bits [13:6] and bits [19:6] of the <REAL> parameter represent the availability mask, the upper memory addressing limits and the lower memory addressing limits respectively. With the MOD2 and MOD1 MCM, bits [3:4], bits [9:6] and bits [15:6] of the <real> parameter represent the availability mask, the upper memory addressing limits and the lower memory addressing limits respectively. The <INTEGER> parameter is the memory module number.

STOP (MACHINEOPS)

STOP generates the HALT operator. If two parameters are specified, the two values are loaded into the A and B registers before the HALT operator is executed. If four parameters are specified, the third and fourth parameters are interpreted as addresses into which the values in the A and B registers are to be stored following execution of the HALT operator.

STOP77 (MACHINEOPS)

The STOP77 intrinsic is similar to the STOP intrinsic, except that the B7000 operator STOP is generated. STOP77 is valid only when the B7000 compiler option is set.

SUSPEND (MACHINEOPS)

The SUSPEND intrinsic is similar to the PAUSE intrinsic, except that the B7000 operator Pause Until Interrupt (PAUS) is generated. SUSPEND is valid only when the B7000 compiler option is set.

THIS INSTRUCTION WILL FUNCTION LIKE AN IDLE INSTRUCTION IF THE PROCESSOR IS IN NORMAL STATE. IF THE PROCESSOR IS IN CONTROL STATE IT WILL IDLE

UNTIL INTERRUPT. WHEN THE INTERRUPT OCCURS IT WILL EXECUTE THE NEXT INSTRUCTION. THUS, IT WILL NOT ENTER HARDWARE INTERRUPT.

TAG (WORD)

TAG is implemented in NEWP as it is in ESPOL.

THIS INTRINSIC WILL ALLOW THE PROGRAMMER ACCESS TO THE TAG PART OF A WORD. FOR EXAMPLE:

```
R:=W.TAG;
W:=0 & 7 TAG;
```

TIMER (MACHINEOPS)

TIMER is implemented in NEWP as it is in ESPOL.

THIS INTRINSIC WILL GENERATE THE SINT INSTRUCTION. THE PARAMETER PASSED TO THE INTRINSIC IS THE AMOUNT OF TIME THE INTERVAL TIMER IS TO BE SET FOR IN UNITS OF 512 MICROSECONDS.

UNLOCK (MISC)

UNLOCK is implemented in NEWP as it is in ESPOL, with the additional restriction that UNLOCK can be used only in CONTROLSTATE blocks.

VECTORCHECKSUM (MISC)

VECTORCHECKSUM(<subscripted variable>,<integer>) returns the checksum calculated over <integer> words starting at the location described by the <subscripted variable>, which must be an element of an INTEGER, REAL, BOOLEAN, or WORD array. VECTORCHECKSUM uses vectormode operators; an INVALID OP interrupt is generated if the processor does not support vectormode.

VIA (REFERENCE)

The syntax "<type> VIA <word primary>" is used to access an item referenced by <word primary> (which is assumed to be a reference generated by the "REFERENCE TO <primary>" syntax) as a value of the specified <type>. <Type> may be any of the following simple types: INTEGER, BOOLEAN, REAL, DOUBLE, POINTER, EVENT, DESCRIPTOR, FILE, TASK, WORD.

EXAMPLES OF THIS INTRINSIC FOLLOW.

```
R:=REAL VIA W;
    VALC(W)
    NAMC(R)
    STOD

R:=REAL VIA WORD AT A[5];
    LT8 5
    NAMC(A)
    INDX
    LODT
    LOAD
    NAMC(R)
    STOD

D:=DESCRIPTOR VIA W;
    NAMC(D)
    NAMC(W)
    LODT
    LOAD
    EXCH
    OVRD
```

VIA PROCEDURE ENTRY (REFERENCE)

The "VIA" reference notation has been extended. In addition to the <type> VIA <word primary> form for simple objects, a similar form applies to procedure invocation:

<procedure name> VIA <word primary>.

The <word primary> is used as a reference to effect procedure entry; it should result in an IRW to a PCW for the desired code the appropriate environment. All type checking and parameter matching is performed according to the declared procedure heading, but the address couple of the procedure is irrelevant.

Examples:

```
T:=PROC VIA REFERENCE TO W (PARAM)
MYGEORGE VIA WORDSPIB[SNR,SIRWTOPALACE] (WHY)
```

WHOAMI (MACHINEOPS)

WHOAMI performs the same function in NEWP as the MYSELF intrinsic does in ESPOL.

THIS INTRINSIC GENERATES THE WHOI INSTRUCTION. THIS INSTRUCTION WILL RETURN THE PROCESSOR NUMBER.

WORD (WORD)

WORD is implemented in NEWP as it is in ESPOL.

ZAP (MACHINEOPS)

ZAP is implemented in NEWP as it is in ESPOL.

THIS INTRINSIC GENERATES THE PTPA INSTRUCTION. THIS INSTRUCTION WILL INTERRUPT ALL PROCESSORS WITH AN ALARM INTERRUPT. IT CAN INTERRUPT A PROCESSOR IN CONTROL STATE.

8 COMPILER CONTROLS

8.1 COMPILER OPTIONS

The compiler options available in NEWP are identical to those in ALGOL, except for the changes, additions, and deletions noted below. The following features differ from the ALGOL semantics:

- Specifying an option without including SET, RESET, or POP causes no action other than setting that option. Specifically, STANDARD OPTIONS ARE NOT RESET (e.g. $MERGE does NOT reset LIST). The CLEAR option, described below, can be used to reset all standard options. Examples:

```
$ MERGE          % IN NEWP HAS THE SAME EFFECT AS:
$ SET MERGE      % IN ALGOL OR NEWP


$ CLEAR MERGE    % IN NEWP HAS THE SAME EFFECT AS:
$ MERGE          % IN ALGOL
```

- The dollar sign ("$") must appear in column 1, 2, or 3.

- Compiler-provided (standard) options are recognized in option expressions (e.g. in the expression "$SET LIST=MYOPTION OR OMIT", OMIT is recognized as the compiler-provided option and is not interpreted as a user

option).

- The options MERGE, LIST, SEPCOMP, and NEW allow a file name to be specified as a string enclosed in quotes (the string may not include the quote character). The specified file name is used to set the TITLE attribute of the compiler files TAPE, LINE, HOST, and NEWTAPE, respectively.

```
$ SET MERGE "(FREDS)ALTERFILE"
```

- The INSTALLATION option accepts either a single intrinsic number or a range of numbers.

- The MAKEHOST option does not allow an <environment-list> to be specified. SEPCOMPable environments in NEWP are controlled by the SEPCOMPLEVEL block direction.

- The SEPCOMP option has different semantics in NEWP than it does in ALGOL. SEPCOMP in NEWP is described in a later section of this document.

- The STATISTICS option produces MCP statistics (as in ESPOL, not ALGOL) for each procedure. Also, statistics will be summarized for any block for which the block direction STATSUMMARY appears.

- Cross-reference generation (XREF) is performed by the NEWP compiler itself and is controlled by two compiler options, XREF and XREFFILES. If XREF is set, a cross-reference listing is produced. If XREFFILES is set, cross-reference files for use by SYSTEM/INTERACTIVEXREF are generated. These options can be set or reset independently of each other.

*See also
8.2 Block Directions
8.3 SEPCOMP

The following compiler options are implemented in NEWP in addition to the standard ALGOL options:

B7000 (RESET if B6000, SET if B7000)

The B7000 option, if SET, specifies that the machine on which the compiler is running is a B7000 Series machine. If RESET, the compilation is assumed to be running on a B6000 Series machine. This option should be explicitly set or reset

because the compiler is machine sensitive.

The heading line on a NEWP compilation listing identified the compiler as either the B7000 or B6000 NEWP compiler, depending on which system it was being run on. It now identifies itself simply as the "LARGE SYSTEMS NEWP" compiler. A line has been added to the summary to indicate for which machine code was generated (depending on the B7000 dollar option). This statement is true only for the NEWP compiler on the B7000 SYSTEM tape. The B6000 NEWP compiler prints the old "B6000 NEWP COMPILER" heading line with no indication of which machine the code was generated for.

The PLDT will be used in place of LODT for code files compiled for a B7000 series machine.

On a B7000 series machine, VALC on a non-indexed descriptor has been changed to NXLV or NXLN. This was done for optimization purposes, i.e., the index operation is considerably faster on B7000 series machines than the VALC.

When the S register (i.e., register #52) is read, NEWP will emit a PUSH before emitting the code for the "read processor register" on a B7000 machine.

The code emitted for the UNLOCK statement on a B7000 machine has been changed to do a B7800 STOREQ purge; i.e., UNLOCK(R) now produces the code: NAMC<R>,ZERO,STON,INCN. This code will be emitted for a B7000 series machine only.

The code emitted for the STOP77 statement has been changed to do a B7700/B7800 purge; i.e., STOP77(R,S) now produces the code: VALC<R>,VALC<S>,LT8<63>,ZERO,SPRR,EXCH,STOP,DLET,DLET. This code will be emitted for a B7000 series machine only.

CLEAR (cannot be SET or RESET)

The CLEAR option resets all compiler-provided, settable options.

LISTO (RESET)

If LISTO is set, all records from the secondary input file (TAPE) that are voided or replaced and all records from the primary input file (CARD) that are omitted will be printed.

LIST1 (RESET)

If LIST1 is set, a listing is produced during the compiler's first pass compiling modules.

MCP (RESET)

MCP is implemented in NEWP as it is in ESPOL.

THE MCP OPTION WILL CAUSE VALUE ARRAYS AND TRUTHSETS TO BE ALLOCATED IN THE PROCEDURE IN WHICH THEY ARE DECLARED. DATA POOLS WILL BE ALLOCATED IN D1. THIS OPTION WILL ALSO CAUSE THE COMPILER TO ALLOCATE OUTER BLOCK STACK CELLS AT D0. WITHOUT THIS OPTION STACK CELLS WOULD START AT D2.

NOCOUNT (RESET)

NOCOUNT is implemented in NEWP as it is in ESPOL.

PROCREF (SET)

If PROCREF is set, each line which references a procedure will be listed with the line number of that procedure's declaration.

READLOCK (RESET)

READLOCK is implemented in NEWP as it is in ESPOL.

IF THIS OPTION IS SET CODE WILL BE GENERATED IN THE BUZZ STATEMENT TO PLACE THE SNR,PIR,PSR AND SDI INTO THE LOCK. THUS, IF A DUMP IS TAKEN ONE CAN TELL WHICH STACK OWNS THE LOCK.

READLOCKTIMEOUT DOLLAR OPTION (B7000)

READLOCKTIMEOUT has been implemented as a dollar option. This option will check to see if hard locks are locked too long.

UNDERLINE (RESET)

If UNDERLINE is set, all procedure names in procedure declarations will be underlined on the output listing.

STANDALONE

A new $ option has been added to the NEWP compiler for use in compiling stand-alone system programs such as the SYSTEM/LOADER. This option is $STANDALONE. To be effective, this option must be set prior to the beginning of the program and $MCP must also be set. As with the MCP itself, the compiler block directive on the outer block must specify into which code segment the outer block is

to be compiled; e.g., SEGMENT=5. In addition, the procedures that must run in control state must be specified as CONTROLSTATE in their block direction.

This option causes the compiler to prepare a complete memory image of the program so that the program is "ready to run" when loaded into the system (starting at memory location zero). The maximum size of this memory image is 22,000 words.

This memory image consists of the following items:

1. The D0 stack image (location (0,9) has been set to zero, this was the LINEINFO dictionary descriptor).

2. All code segments.

3. All value arrays.

4. All "pool data" items.

5. The storage space for all SAVE arrays declared at a D0 location.

6. The proper entry PCW for the outer block at (0,3).

7. A "memory" descriptor at (0,4).

The memory image does NOT include:

1. Allocated data storage for any array declared within a procedure or any D0 array not declared to be SAVE. (No errors or warnings are given for these, as it is presumed that a proper presence- bit handling routine will be provided by the user.)

2. No "memory links" are provided to separate any of the allocated storage areas.

The code file generated by the NEWP compiler when the $STANDALONE option is set contains only three items:

1. A bootstrap in code segment 0.

2. A valid "SEG0" in code segment 1.

3. The memory image as described above, starting in code segment 2 and continuing to the end of the file. The SEG0[18] word (the segment dictionary pointer) properly describes the D0 image in the first part of the complete memory image.

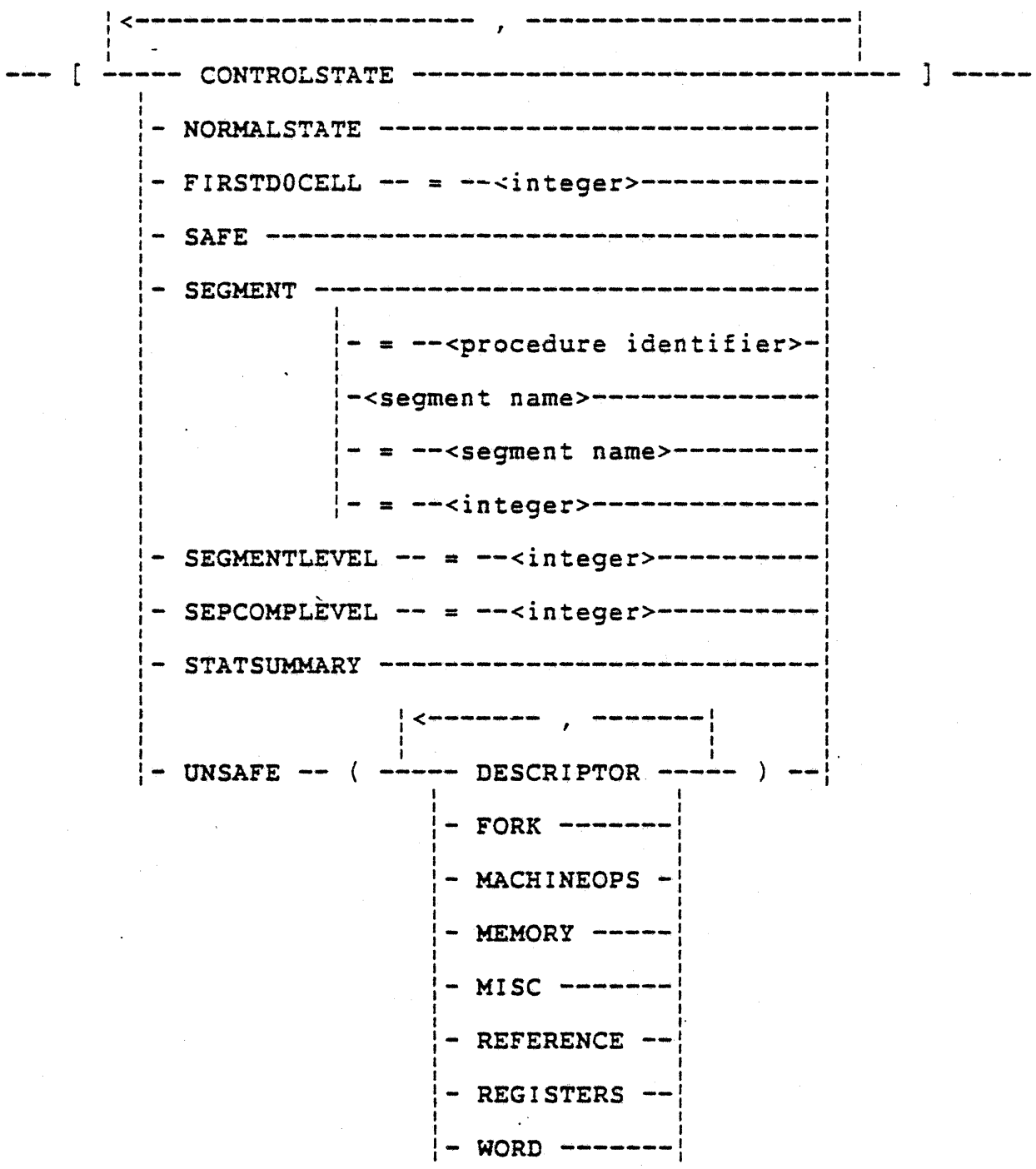
The code file does NOT contain any SEPCOMP information, no BINDINFO no LINEINFO, and no PPB.

If $STACK is set, the $STANDALONE option will print a table of segment descriptors and data descriptors in D0, indicating which ones remain absent and showing the memory locations of the ones made present.

The following ALGOL compiler options are not implemented in NEWP:

AREACLASS
AUTOBIND
BCL
BEGINSEGMENT
BIND
BINDER
BREAKHOST
BREAKPOINT
B7700
CHECK
DOUBLESPACE
DUMPINFO
ENDSEGMENT
EXTERNAL
FORMAT
GO
GO TO
HOST
INITIALIZE
INTRINSICS
LEVEL
LIBRARY
LISTDELETED
LISTOMITTED
LOADINFO
NOSTACKARRAYS
NOXREFLIST
OPTIMIZE
PURGE
SEGDESCABOVE
SEQERR
STOP
USE
WRITEAFTER
XDECS
XREFS

8.2 BLOCK DIRECTIONS


CONTROLSTATE
NORMALSTATE
FIRSTDOCELL -- = --<integer>
SAFE
SEGMENT
= --<procedure identifier>
<segment name>
= --<segment name>
= --<integer>
SEGMENTLEVEL -- = --<integer>
SEPCOMPLEVEL -- = --<integer>
STATSUMMARY
UNSAFE -- (DESCRIPTOR) --
FORK
MACHINEOPS
MEMORY
MISC
REFERENCE --
REGISTERS --
WORD

Example:

```
BEGIN [SEGMENT OUTERBLOCK, SEPCOMPLEVEL=5]
  ...
  PROCEDURE P(X);
  VALUE X;  REAL X;
  BEGIN
   [UNSAFE(MEMORY,WORD),SEGMENT=5,CONTROLSTATE]
   ...
  END P;
```

```
  ...
END.
```

Through the use of block directions, the programmer can control segmentation, the use of potentially dangerous constructs, and other compilation details. Block directions may occur inside brackets immediately after any BEGIN. In order to make the following discussion more readable, the word "block" has been used for the concept of "block or procedure".

In general, block directions are inherited by nested blocks unless overridden by block directions appearing in the nested block. However, the UNSAFE block direction is not inherited. TEMPORARY -- Temporarily, UNSAFE is inherited by nested blocks.

The following keywords are recognized as directions to the compiler:

FIRSTFREED0CELL

> The default value of the FIRSTFREED0CELL block direction is now 10 (decimal) rather than 224 (decimal). Furthermore, FIRSTFREED0CELL is prevented from having a value less than 10 (decimal). This causes no problem with the MCP, as its symbolic has an explicit setting for this block direction.
>
> In addition, a warning will now be issued when FIRSTFREED0CELL has a value less than 222 (decimal) and the dollar option STATISTICS is set. (The compiler sets the value of MCPHIGHSTATNUM=(0,222) to the number of statistic entries when $STATISTICS is set and FIRSTFREED0CELL is greater than 222).

CONTROLSTATE

> The block is to run in control-state. (UNSAFE only)
>
> Control state blocks are now available in the NEWP language. The "unsafe" block directions CONTROLSTATE and NORMALSTATE are now allowed following any BEGIN, specifying that the compound statement is to be a block which is run in control state or normal state, respectively. This concept allows the static specification of normal state versus control state in a NEWP program like the MCP. Previously, NEWP programs used the DISALLOW and ALLOW statements (inherited from ESPOL) to dynamically alter the "control" state of the processor. ALLOW and DISALLOW have been de-implemented as NEWP language constructs.

In the absence of any explicit compiler direction, the "state" of a block (and the body of an ON declaration) is inherited from the containing block; the outer block and all procedures are by default normal state. (The Mark 31 NEWP compiler caused procedures without these explicit state directions to inherit the state from their containing environment and caused all ON declarations to be normal state.)

The explicit use of CONTROLSTATE or NORMALSTATE block direction creates a true block; in particular, a GOTO from outside the range of such a block cannot transfer into the body of the block. Also, a GOTO which transfers from within such a block to a more global block is allowed, but is treated as a "bad-go-to". All "bad-go-to"s cause an effective loss of control, even if both the GOTO statement and the destination label are in (different) "control" state blocks.

The BUZZ and UNLOCK constructs remain in the NEWP language, but compiler now restricts their use to only control state environments; a syntax error is given for their use in normal state.

In addition, compound statements which specify the STATSUMMARY compiler direction are now also treated as distinct blocks.

NORMALSTATE

The block is to run in normal-state.

SAFE (TEMPORARY) Generate syntax errors for all UNSAFE constructs used in this block, except those enabled by subsequent UNSAFE specifications.

SEGMENT

Segmentation information for the block may be specified by one of the following phrases:

a. SEGMENT -- Make a new segment.

b. SEGMENT = <procedure identifier> -- Add the code for this block to the segment which contains the specified procedure. The <procedure identifier> must have appeared previously in a <procedure declaration> or a <forward procedure declaration>.

c. SEGMENT <segment name> -- Make a new segment and associate the specified <segment name> with it.

d. SEGMENT = <segment name> -- Add the code for this

block to the segment associated with this segment name. The <segment name> must have appeared previously in a "SEGMENT <segment name>" compiler direction.

e. SEGMENT = <integer> -- Add to segment <integer>. (UNSAFE only)

SEGMENTLEVEL

The SEGMENTLEVEL block direction must be followed by an integer which specifies the lex level at or below which segmentation takes place.

SEPCOMPLEVEL

The SEPCOMPLEVEL block direction must be followed by an integer which specifies the lex level at or below which separate compilation can occur for declarations.

STATSUMMARY

If the compiler option STATISTICS is set, statistics will be summarized for the block(s) on which the STATSUMMARY block direction appears (statistics are automatically summarized for each procedure when STATISTICS is set).

UNSAFE

Allow the use of the potentially dangerous constructs specified by the following keywords:

DESCRIPTOR

Allow the DESCRIPTOR data type and DESCRIPTOR type transfer function.

FORK

Allow FORK statements.

MACHINEOPS

Allow the use of the following intrinsics: EVAL (temporary), EXIT, FAILREGISTER, FMMRREADLOCK, HEYOU, IGNOREPARITY, INTERRUPTCHANNEL, MAKEPCW (temporary), MOVESTACK, PAUSE, RETURN, SCANIN, SCANOUT, SETINHIBIT, SETLIMITS, STOP, STOP77, SUSPEND, TIMER, WHOAMI, ZAP.

MEMORY

Allow the MEMORY array intrinsic.

MISC

Allow address equation, NULL as a <procedure body>, <option list>s on WAIT and WAITANDRESET statements, the OVERWRITE option on the REPLACE statement, SAVE ARRAY declarations, and the following intrinsics: BUZZ, BUZZ47, LEXOFFSET, UNLOCK.

REFERENCE

Allow "REFERENCE TO <primary>", "<type> VIA <word primary>", "<type> AT <address primary>" syntax.

REGISTERS

Allow the REGISTERS array intrinsic and D-register override clause ("@ [<exp>]").

WORD

Allow WORD data type, WORD type transfer function, POINTER(<word exp>), and TAG.

*See also
3 Program Structure
7 UNSAFE Mode

8.3 SEPCOMP

Separate Compilation (SEPCOMP) is implemented in the NEWP compiler itself and does not require invocation of the BINDER. In order to use the SEPCOMP facility, a "host" must be generated by compiling a program with the compiler option MAKEHOST set, causing the compiler to place information necessary for SEPCOMP in the codefile. Patches to the host symbolic can be compiled with SEPCOMP set, resulting in an abbreviated compilation, since only the affected areas of the program are actually compiled.

A SEPCOMP "region" is any declaration that occurs at a lex level less than or equal to the SEPCOMPLEVEL (either the one specified by the SEPCOMPLEVEL block direction or the default). The executable statements of a procedure (or outer block) are considered to be a "text" region. A region can be changed, added, or deleted during SEPCOMP. The text region of a procedure (or outer block) is recompiled if any declaration region associated with it is separately compiled.

At the present time, the programmer is responsible for observing the following restrictions:

a) Do not change a VALUE ARRAY, ARRAY, DEFINE, FORMAT, FILE, TRANSLATETABLE, or TRUTHSET declaration without explicitly causing all references to the changed identifier to be recompiled.

b) Do not patch (via SEPCOMP) COMMENTs and areas of the symbolic which were omitted because the compiler option OMIT was set.

c) Do not change the starting or ending sequence number of a region.

d) Do not put multiple regions on one line.

These restrictions do not apply to patches to declarations or text in procedures above the SEPCOMPLEVEL, since a patch to such a procedure causes the entire procedure to be recompiled.

The user compiler options that were set when the host was compiled and the VERSION that was supplied for the host are preserved and reinstated during a SEPCOMP.

The title of the symbolic file which was compiled as the host is saved in the codefile generated during a MAKEHOST or SEPCOMP compile. This title is used as the default title for the TAPE file when SEPCOMPing (but may be overridden by label-equation) and is selected as follows: if MAKEHOST is set, the titles of the NEWTAPE, TAPE, and CARD (if it is a disk file) files are examined in order, and the first valid title is saved; if SEPCOMP is set, the title of the TAPE file is saved.

*See also
8.1 Compiler Options
8.2 Block Directions

9 ALGOL FEATURES NOT IMPLEMENTED IN NEWP

The following ALGOL features are not implemented in NEWP. Some of these features were considered undesirable or inappropriate in the context of NEWP, while others have been or will be replaced by NEWP features which perform approximately the same function. Page numbers refer to the ALGOL Language Reference Manual.

DECLARATIONS

- The ALPHA declaration, page 4-2, and ALPHA as a <type> in

the <array declaration>, page 4-3.

- OWN variables (OWN as the <local or own> part), page 4-2.
- <Array row equivalence>, page 4-5.
- The DUMP declaration, page 4-13.
- <In-out part>s in FORMAT declarations, page 4-20.
- Non-EBCDIC <simple string>s in formats, page 4-24.
- The O format phrase, page 4-36, and the R format phrase, page 4-37.
- <Forward interrupt declaration>s and <forward switch declaration>s, page 4-42.
- The INTERRUPT declaration, page 4-44.
- The LIST declaration, page 4-46.
- The MONITOR declaration, page 4-48.
- The PICTURE declaration, page 4-51.
- <Parameter delimiter>s other than comma (","), page 4-55.
- Labels and formats as formal parameters (LABEL and FORMAT as <specifier>s in <formal parameter part>s), page 4-55.
- <Switch declaration>s, page 4-60.

STATEMENTS

- The ACCEPT statement, page 5-2.
- The update replacement (":=*") construct for assignment to task or file attributes, page 5-6.
- The ATTACH statement, page 5-11.
- The BREAKPOINT statement, page 5-12.
- The CALL statement, page 5-13.
- The CHANGEFILE statement, page 5-19.
- The CHECKPOINT statement, page 5-20.
- The CONTINUE statement, page 5-30.
- The DEALLOCATE statement, page 5-31.

- The DETACH statement, page 5-32.
- The DISABLE statement, page 5-33.
- The ENABLE statement, page 5-36.
- The EXCHANGE statement (for disk file rows), page 5-38.
- The FILL statement, page 5-39.
- The MERGE statement, page 5-56.
- The multiple (file) attribute assignment statement, page 5-57.
- The ON statement, page 5-58.
- The PROCESS statement, page 5-62.
- Core-to-core I/O (i.e. <core-to-core part>s as <file part>s), page 5-66.
- Formatted I/O (i.e. <format designator>s or in-line editing specifications in <format and list part>s), page 5-66.
- Binary I/O (i.e. "*" as the format in a <format and list part>), page 5-66.
- Free-field I/O (i.e. <free field part>s in <format and list part>s), page 5-66.
- The WHILE clause for FOR-loops occurring in the <format and list part>s of READ and WRITE statements, page 5-66.
- The update replacement (":=*") construct for assignment to the control variable of a FOR-loop occurring in the <format and list part> of a READ or WRITE statement, page 5-66.
- The REMOVEFILE statement, page 5-77.
- All intrinsic <translate table>s that refer to BCL (these translate tables can be declared, however), page 5-79.
- The <replace family-change statement>, page 5-88.
- The REWIND statement, page 5-92.
- The RUN statement, page 5-93.
- The SPACE statement, page 5-104.
- The SWAP statement, page 5-108.

- The VECTORMODE statement, page 5-111.

- The WAIT statement with no parameters (wait for interrupt), page 5-114.

- The WHEN statement, page 5-117.

EXPRESSIONS

- Vertical bar ("¦") as the logical operator OR in <Boolean term>, page 6-9.

- <Designational expression>s, except <label identifier>, page 6-16.

- The following intrinsics (page 6-19 to 6-30):

  ARCCOS, ARCSIN, ARCTAN, ARCTAN2, ATANH
  CHECKSUM, COS, COSH, COTAN
  DABS, DARCCOS, DARCSIN, DARCTAN, DARCTAN2, DCOS,
  DCOSH, DELTA, DEF, DEFC, DEXP, DGAMMA, DIMP, DINTEGER,
  DLGAMMA, DLN, DLOG, DMAX, DMIN, DNABS, DSCALELEFT,
  DSCALERIGHT, DSIN, DSINH, DSQURT, DTAN, DTANH
  ENTIER, ERF, ERFC, EXP
  GAMMA
  LINENUMBER, LNGAMMA, LOG
  NORMALIZE
  RANDOM
  SCALELEFT, SCALERIGHT, SCALERIGHTF, SCALERIGHTT, SIGN,
  SIN, SINH, SIZE(<pointer identifier>), SQRT
  TAN, TANH

MISCELLANEOUS

- Identifiers, numbers, and strings continued across card images.

- Multi-character operators with embedded blanks. As an exception, the update replacement operator, ":=*", is allowed to have one blank between the "=" and the "*" (i.e. ":= *" is allowed).

- <Global part> in a program unit, page 3-1.

- KIND=READER for the compiler file CARD (CARD must be label-equated to a disk file), Appendix E (of the ALGOL Language Reference Manual).

- Batch Facility, Appendix F (of the ALGOL Language Reference Manual).

- BDMS statements (see B7000/B6000 Series DMSII HOST Reference Manual, #5001498).

D2430 MCP - "MCP" RESTRUCTURING

Several changes have been made to improve the maintainability of th MCP on the III.1 release, as follows:

The MCP has been grouped into logical modules. Each global ite belongs to a particular module. Various "interfacing" define have been added for accessing data from other modules.

Several external procedures which used MCP data structures hav been moved into the MCP symbolic. These include SORT MAINTENANCE, Reader/Sorter routines and some of the intrinsic which had previously been written in ESPOL.

These changes have affected the way in which an MCP is generated, a follows:

Compilation in NEWP

- DUMPINFO and LOADINFO are not supported.
- Dollars cards not beginning with SET or RESET will not affec unmentioned options. To reset all options, $CLEAR can b used.
- The compiler produces its own cross-reference SYSTEM/XREFANALYZER need not be run.
- A MAKEHOST and a SEPCOMP facility exist similar to ALGOL except that NEWP does its own binding.

Symbolic Consolidation

- SORT and MAINTENANCE need not, and cannot, be separatel compiled and bound. They can, however, be changed usin SEPCOMP.
- The DCALGOL components of the MCP are compiled and bound a before.
- The ESPOL and PLI intrinsics no longer do a LOADINFO from th MCP.

10 NEWP PROGRAMMING PROBLEMS

THE FOLLOWING IS A LIST OF PROBLEMS WHICH COULD BE ENCOUNTERED

WHILE PROGRAMMING IN NEWP.

CALLS ON MCP PROCEDURES:

WHEN WRITING STAND ALONE NEWP PROGRAMS, ONE MUST USE CARE NOT TO CALL MCP ROUTINES UNLESS THEY HAVE BEEN WRITTEN. FOR EXAMPLE THE FOLLOWING CONSTRUCTS CALL MCP ROUTINES:

```
WAIT
PBIT ON ANYTHING
EXIT A PROCEDURE WITH AN ARRAY DECLARED (BLOCKEXIT)
FORK
HARDWARE INTERRUPT
STACK VECTOR
ARRAYDEC FOR TWO DIMENSIONAL ARRAYS
```

ARRAY LOST IN BLOCKEXIT:

```
BEGIN [UNSAFE (MISC,WORD,DESCRIPTOR)]
 PROCEDURE P(W);
 VALUE W;
 WORD W;
 BEGIN
    ARRAY A = W [0];
    A[0]:=0; % MAKE PRESENT
 END; % BLOCKEXIT WILL FORGET ARRAY
 PROCEDURE P2;
 BEGIN
    ARRAY A[0:14];
    WORD WA = A;
    A[0]:=0;
    P(WA); % PASS THE MOM
 END;
 P2;
END.
```

UP STACK MOM:

```
BEGIN [UNSAFE (MISC,WORD,DESCRIPTOR,MACHINEOPS)]
 PROCEDURE P3(W);
 VALUE W;
 WORD W;
 BEGIN
    ARRAY A = W [0];
    A[0]:=0; % MAKE PRESENT
    EXIT;
 END;
 PROCEDURE P4;
```

```
 BEGIN
    ARRAY A[0:14];
    WORD WA = A;
    P3(WA); % PASS THE MOM
    A[0]:=0; % PBIT AGAIN
 END;
 P4;
END.
```

READ ONLY BIT WILL PROTECT DATA:

IF THE MEMORY PROTECT BIT IS ON IN A DATA DESCRIPTOR ONE CAN NOT DO A STORE OPERATION USING THE DESCRIPTOR. USING THE OVRD OPERATOR WILL NOT ALLOW THE WORD TO BE STORED.

IRW LOOP:

```
BEGIN [UNSAFE (WORD,REFERENCE,MISC)]
 WORD W;
 REAL R = W,X;
 W:=REFERENCE TO R;
 X:=R;
END.
```

SECTION 3

SYSTEM INITIALIZATION

INTRODUCTION

This section discusses the MINILOADER, SYSTEM/LOADER and some MCPHOST procedures. Two MCPHOST procedures, GETITGOING and PRIMARYINITIALIZE, are responsible for nearly all system initialization.

B 7000 SYSTEM INITIALIZATION

B 7000 system initialization consist of bringing the master control program into memory and placing this program in control of the system. To accomplish this, one of two paths must be traversed as shown in figure 1-17. The path on the right of the figure assumes a valid MCP is on disk and if this is the case a halt/load may be performed. If a valid MCP is not on disk or there is no valid directory on disk, then the left path must be traversed first, followed by entry into the disk boot.

Progression through the boots shows a continuing increase in sophistication. At the hardware level, code is brought into memory by a very simple algorithm: if a card load is being performed, cards from the selected card reader will be read until a bad result descriptor is returned (usually forced with an invalid character in card 9). If a disk load is being performed, 8192 words will be read from the selected disk into memory. If there are any unexpected errors in either of these hardware functions the operator must start the operation from the beginning. Once the nine-card loader takes control of the system, cards from the unit the loader itself was read from may continuously be read. Unexpected read checks, etc. are handled with no problem.

The deck of cards normally read into memory by the 9-CARD LOADER is the MINILOADER. This program, written in ESPOL, can read into memory any given file from any specified tape unit as long as that unit is connected to the halt/load IOM. Tape parity errors are not retried.

The MINILOADER usually reads in the SYSTEM/LOADER. This is the first major program to be run on the B7000 and is responsible for constructing the directory, loading the specified MCP from tape to disk, and entering the MCP via the disk boot.

MINILOADER

SYSTEM/MINILOADER is read into memory by the 9-CARD LOADER and its purpose is to read a specified file from a given tape unit and then to enter this code. The MINILOADER's segment dictionary is read into memory starting at address 0 and its code will be above the dictionary but below address 4"1F00.". A detailed description of tis process and pictures of the stack can be found n the Software Operational Guide, Volume 2, Section 2 (form number 5001788). One of the first things the MINILOADER does is to increment the S register by 16 thus taking advantage of the IOCB, HOME ADDRESS location, etc. already set up by the 9-CARD LOADER. After the S register has been incremented the MINILOADER follows the basic outline given on the following pages.

Shown below is a list of the six procedures declared in the MINILOADER:

1. JUSTEXIT
2. IO (CHNL, IOCW, CDL)
3. READTAPE
4. JUMPTM (HOWMANY)
5. HI (P1, P2)
6. READINTHECODE

MINILOADER OUTLINE

1. Read in the parameter card to determine where the tape unit is and what code file is required. A typical parameter card is: 1/20 SYSTEM/LOADER

2. Store the channel and unit number info from the parameter card and then build a STANDARD FORM NAME. The STANDARD FORM NAME of our typical file is:

   48"11010206", 8"SYSTEM", 48"06", 8"LOADER",

   The STANDARD FORM NAME is described as follows:

   CHAR. 1. Total number of characters in the whole string

(self-inclusive).

CHAR. 2. Security bytes:

0. For MCP use.

1. User file.

2. System file.

3. Usercode specified.

CHAR. 3. Number of identifiers in the file name

CHAR. 4. Identifiers, each preceeded by one character giving the length of that identifier (not self-inclusive).

3. Rewind the tape just in case it needs to be rewound.

4. Space up the tape one tape mark. This will place the tape read head immediately before the directory. If this step is researched in detail in the miniloader it will not be understood unless it is realized that when tape is spaced, a zero in the CDL field indicating number of blocks to be spaced will be interpreted by the hardware as a space of 100. The spacing will stop when an error occurs or a tape mark is found.

5. Read in the tape directory. The format of a LIBRARY TAPE is shown in figure 3-1. The directory consists of one or more blocks, each 901 words long consisting of a one word transaction count and one word of link information used only for multi-reel library tapes. The link information is followed by up to 899 words of file names in standard form. All blocks are completely packed and a file name may be split across directory blocks. The end of the directory is flagged by character 1 following the last standard form name having a value of 0. The first word in all blocks is a transaction number. In the tape directory, the number in the first block is minus one (-1). In each succeeding block, the number is decremented by one. When the MINILOADER reads the tape directory it will check the transaction number and if it is equal to or greater than 0 the tape will not be used.

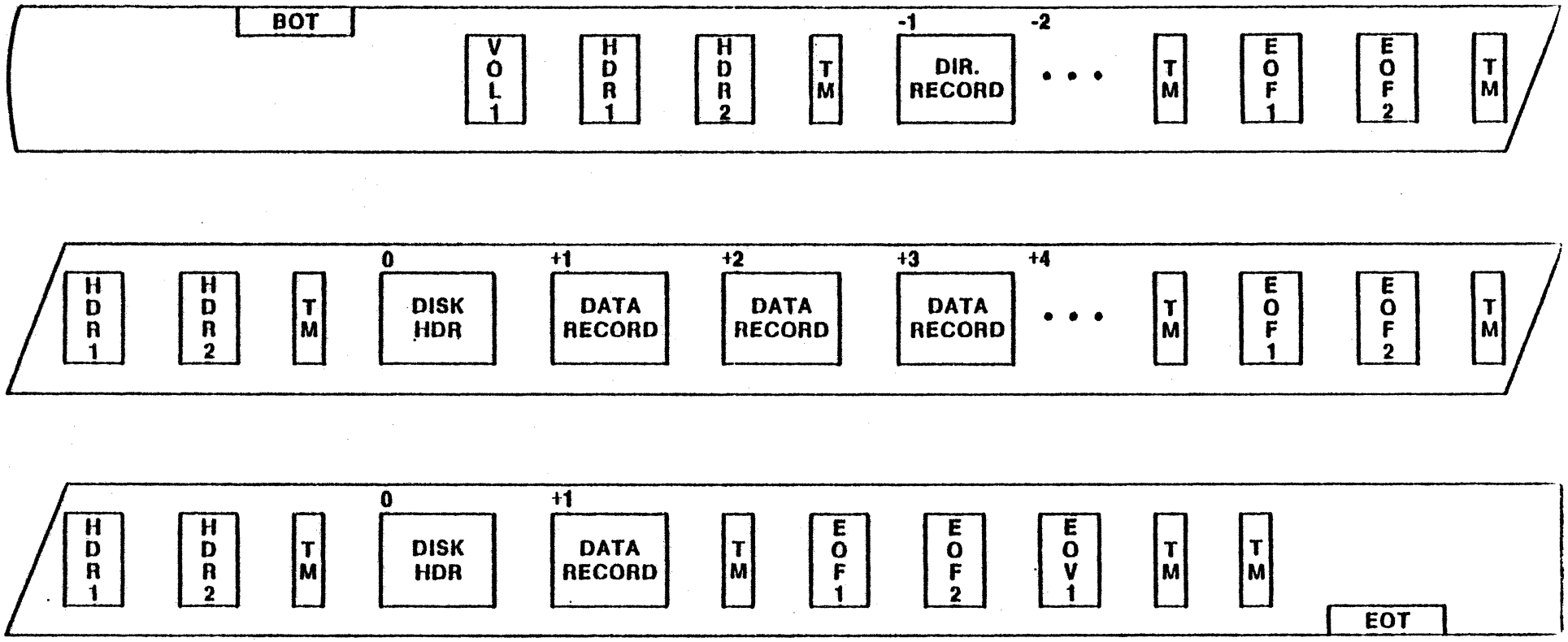

BOT
VOL 1
HDR 1
HDR 2
TM
-1
DIR. RECORD
-2
• • •
TM
EOF 1
EOF 2
TM
HDR 1
HDR 2
TM
0
DISK HDR
+1
DATA RECORD
+2
DATA RECORD
+3
DATA RECORD
+4
• • •
TM
EOF 1
EOF 2
TM
HDR 1
HDR 2
TM
0
DISK HDR
+1
DATA RECORD
TM
EOF 1
EOF 2
EOV 1
TM
TM
EOT

Figure 3-1. Library Tape

6. Find the required file's location on the tape by use of the directory then space up to the file. For each standard form name checked in the directory, 3 tape marks will have to be jumped.

7. Read in the file's header. The header is the first block in the file. The header and all other blocks will contain one extra word (the first word) used as a transaction count. For non-directory blocks, the first word will contain a 0. Each succeeding block will have this first word incremented by 1.

8. Read in the code.

   a. Now that the header has been read in, it is time to read in the code file. The code file was previously written to tape a row at a time with each block containing 30 records each 1 disk segment long (30 words with tags in line).

   b. The MINILOADER does not read the entire code file into memory. Only the save information is read in, but to do this two things must be known:

      1) The location of the save stuff on tape.

      2) The number of words to be read.

   c. The information required here may be obtained from SEGO of the program's code file and since it is written to tape first, it is now read into memory (into IOAREA). The length of the code file is placed in variable K and the segment address is placed in variable CODESTART.

   d. Now that the save stuff's length and location have been determined we space up the tape to where the information is located and start reading it in. This info is read into IOAREA without tag transfer and then it is moved to the proper location in memory (starting at 4"2100") with tags in place.

9. Rewind the tape.

10. Call JUSTEXIT. This call is made to clear the B 6000 recursive interrupt counter (similar to the B 7000 control mode register) and serves no purpose on the B 7000 system.

11. Move the working stack immediately above the loaded code and adjust the F, S and D[1] registers.

12. Change the IOM's home address register to point into the new stack area.

13. Move the save information down to address 0.

14. Go to the PCW at 0,3. This will be absolute address 3 and will contain the PCW that points to the first instruction in the loaded program's outer block.

SYSTEM/LOADER

SYSTEM/LOADER is a program with many abilities. The LOADER is normally brought into memory, by the MINILOADER from a tape named SYSTEM and once the LOADER begins execution it starts reading cards to determine its objective(s). Among the many possible actions the LOADER may take are listing input cards, taking memory dumps, loading an MCP to disk, constructing the FLAT DIRECTORY, etc.

When the LOADER is first exited into from the MINILOADER a few instructions are executed that move the code and stack to a better location and then call a procedure named OUTERBLOCK. This procedure is not the actual outer block of the LOADER but it might as well be because it is the base of this program.

Figure 3-2 illustrates the flow of the SYSTEM/LOADER. Outline of the LOADER is detailed on subsequent pages.

Figure 3-2. COLD START Flow of SYSTEM/LOADER

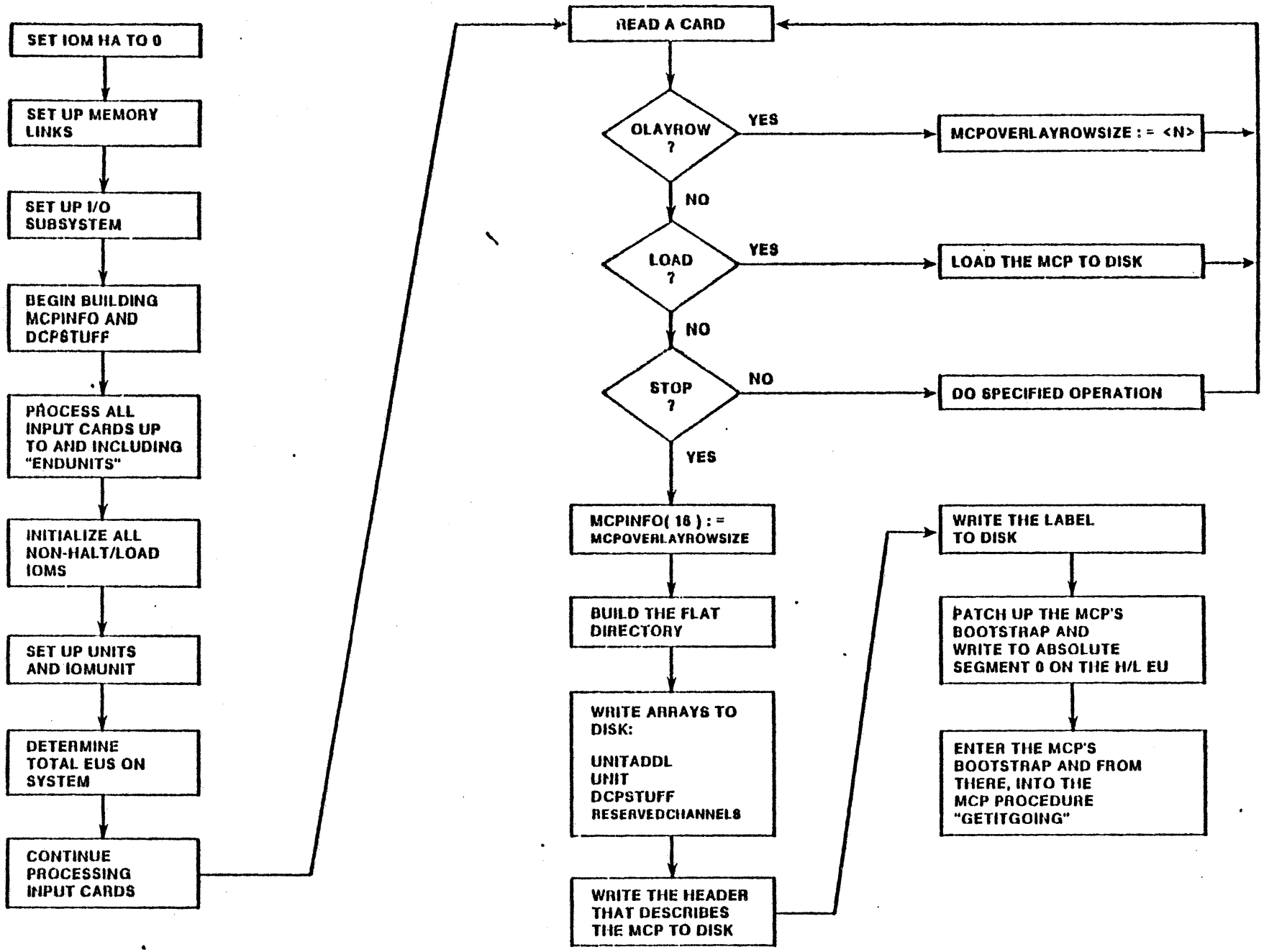


Figure 3-2. Cold Start Flow of SYSTEM/LOADER

OUTLINE OF SYSTEM/LOADER

1. Move code up in memory.

2. Move stack down where code was.

3. Fix up registers to point to stack.

4. Fix up segment descriptors to point to new code area.

5. Call procedure OUTERBLOCK.

6. Call INITIALIZEIOM to set the Halt Load IOM's Home Address Register to zero.

7. Call INITMEMORY to set up memory links around available memory areas.

8. Call PERIPHERALINIT to set up enough of the I/O subsystem to process input cards. This procedure will do the following:

   a. Allocate memory for IOQUEUE, IOMUNIT, RESULTQ, IOMHOMEADDR, and FAILIOCBS.

   b. Make IOM point to these memory areas.

   c. Test channels to find ODT, card reader, and line printer.

9. Call INITIALIZESTUFF to display the LOADER BOJ message on the ODT. In addition, this procedure will begin building the MCPINFO array.

10. Call PROCESSINPUTCARDS to process input cards to and including the "ENDUNITS" card. That is, this procedure will process the B7000 unit cards.

11. Call INITIALIZESLAVEIOMS to set up all non-Halt/Load IOM's IOQUEUE, IOMUNIT, RESULTQ, and IOMHOMEADDR registers.

12. Call WHATSOUTTHERE to see what peripherals are on the system.

13. Call SELECTOR to process the remaining cards. This procedure will read and pocess all cards until a "STOP" card is fond. When the "STOP" card is read, the following is done:

    a. Build the flat directory.

    b. Write the bootstrap on the Halt/Load unit at address 0.

c. Put a copy of the bootstrap at memory address 0.

d. Transfer control to the bootstrap word 8.

14. The bootstrap will set up a few registers and read in some of the MCP. It will then begin executing the MCP procedure GETITGOING.

GETITGOING

GETITGOING is the first MCP procedure entered from the disk boot and is responsible for bringing MCPINFO, UNITCARDS, etc. into memory. This procedure also brings in the MCP's segment dictionary and save code and updates several locations in the dictionary. PRIMARYINITIALIZE is called to continue the initialization process. GETITGOING is never returned to after the call on PRIMARYINITIALIZE

EXECUTION OF GETITGOING

GETITGOING is passed the SYNC I/O home address command as a parameter by the disk boot. This parameter is deleted by GETITGOING. GETITGOING also expects the following to already be in memory or set up in some manner:

S 4"1E00"

ALL IOM HA registers 0

M[2] set up. See word 2 in the disk boot.

M[3] The PCW for GETITGOING

M[4] hardload result descriptor.

M[30] 4"F9999F0F999F"

M[31] 6"COLD STA"

M[32] 6"RTING "

M[16381] 4"F9999999999F"

M[16382] 6"COLD STA"

M[16383] 6"RTING"

OUTLINE OF GETITGOING

1. Enter GETITGOING from the disk boot. Memory will have the following

A. Bootstrap.

B. Open area.

C. GETGOING read in by bootstrap.

D. Open area.

E. Halt load processors stack.

2. Pick up information left by bootstrap.

3. Build MEMORY descriptor.

4. Set up hardware interrupt based on machine type.

5. Check to see if a memory dump is requested. If a dump is requested it will be taken and initialization will start over.

6. Set up an initialization data area and code area above MOD one. This area is used to hold arrays and code used during initialization.

7. Move GETITGOING to the initialization code area.

8. Move H/L processors stack to initialization data area.

9. Read pack label from halt load unit.

10. Use pointer in pack label to find FLAT directory header. Read in the header (FLAT FLAT). Also read in segment zero of FLAT directory.

11. Use pointer in segment zero to find MCP structure table. Read in structure table which is a set of pointers (record numbers in FLAT) to structures such as MCPINFO.

12. Read in configuration information. Pointed to by structure table.

13. Read in MCPINFO.

14. Read in MCP code file disk header for this H/L unit. The header is pointed to by MCPINFO.

15. Read in segment zero of the code file.

16. Read in D0 stack image from MCP code file to memory at D0SETTING.

17. Set the D0 register to D0SETTING.

18. Place information generated so far in the D0 stack.

19. Read in other structures (UNITADDL, SWAPPERINFO, FUNCTIONTABLE).

20. Read in PCTABLE (B7000 peripheral table).

21. Set up a LOCAL code memory area at low order memory for save procedures. Set up a RESIDENT code area above the D0 stack for save procedures.

22. Read MCP procedures into the areas set up for save procedures. Invoke some initialization procedures to set up the software environment based on machine type. In addition, some of the procedures read in are based on machine type.

23. Some information to be passed on to PRIMARYINITIALIZE is placed in an array.

24. Call PRIMARYINITIALIZE passing the array of information as a parameter.

PRIMARYINITIALIZE

PRIMARYINITIALIZE is called by GETITGOING and is the major MCP procedure involved in the initialization process. It should be pointed out, however, that the procedure SECONDARYINITIALIZE local to PRIMARYINITIALIZE does most of the work.

OUTLINE OF PRIMARYINITIALIZE

1. Make D0+3 point to A hardware interrupt routine in PRIMARYINITIALIZE.

2. Determine if system is Tightly Coupled (TC) from configuration file.

3. Generate IOM and CPM masks from configuration file.

4. All IOM's are told to load home address register.

5. All slave (non-H/L) CPM's are started. The slave CPM's will execute the procedure called SLAVECPMHOLDER. In this procedure the H/L CPM can give them commands.

6. The H/L CPM sets its CMR register and tells the saves to set their CMR register.

7. All CPM's set time of day register.

8. Set up Halt/Load stacks. These stacks are set up with an environment which will allow the CPM's to enter SECONDARYINITIALIZE. See figure 3-3.

9. The initial PIBDESC is set up.

10. A STACKINFO and STACKSTATUS array is set up. This is a dummy array at this point.

11. The BOXINFO array is initialized.

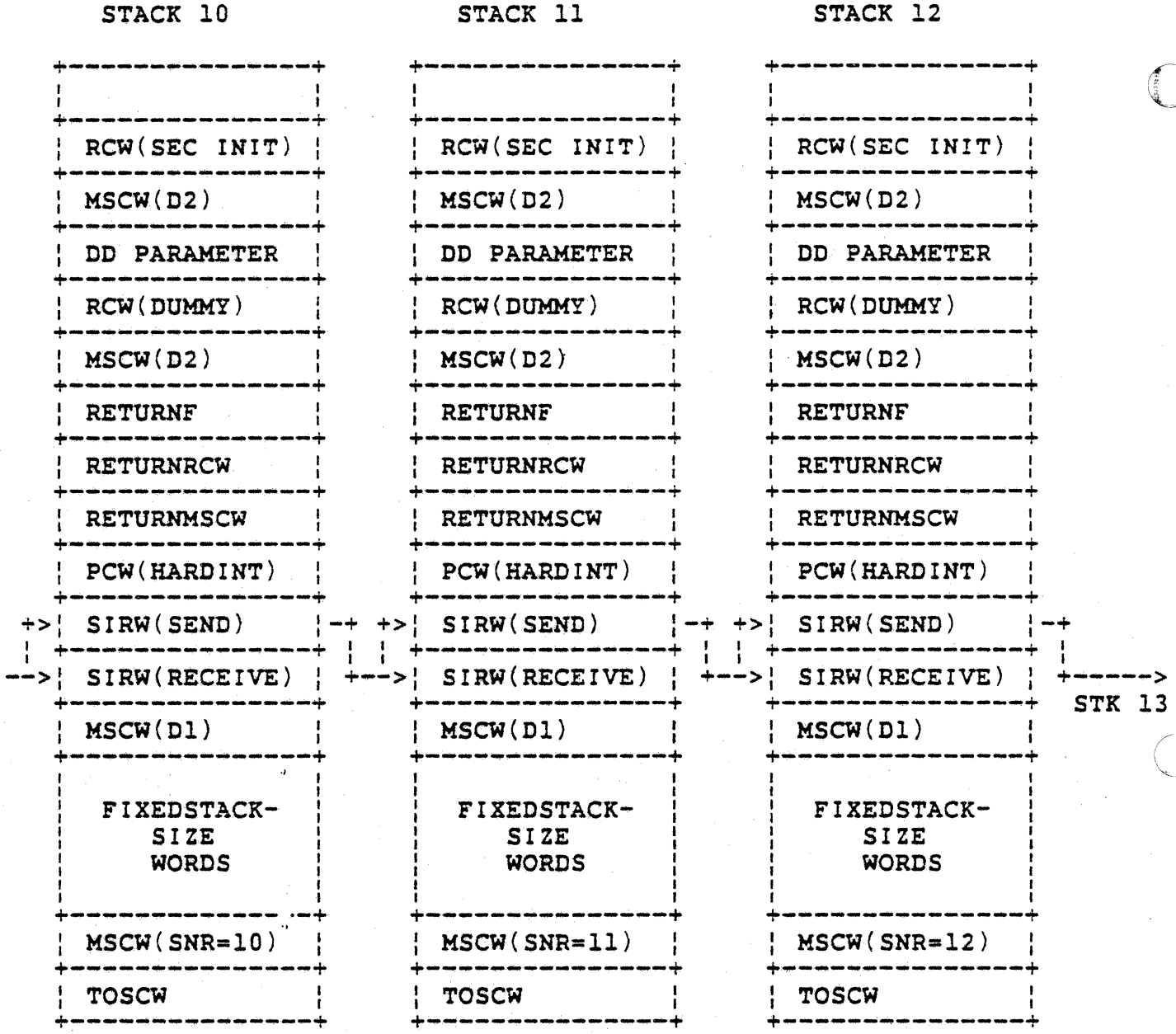
12. The slave CPM's are told to invoke BOOTIT which they do.

13. The H/L CPM invokes BOOTIT.

OUTLINE OF BOOTIT

1. Move to proper Halt Load stack. The stack number is based on the CPM number.

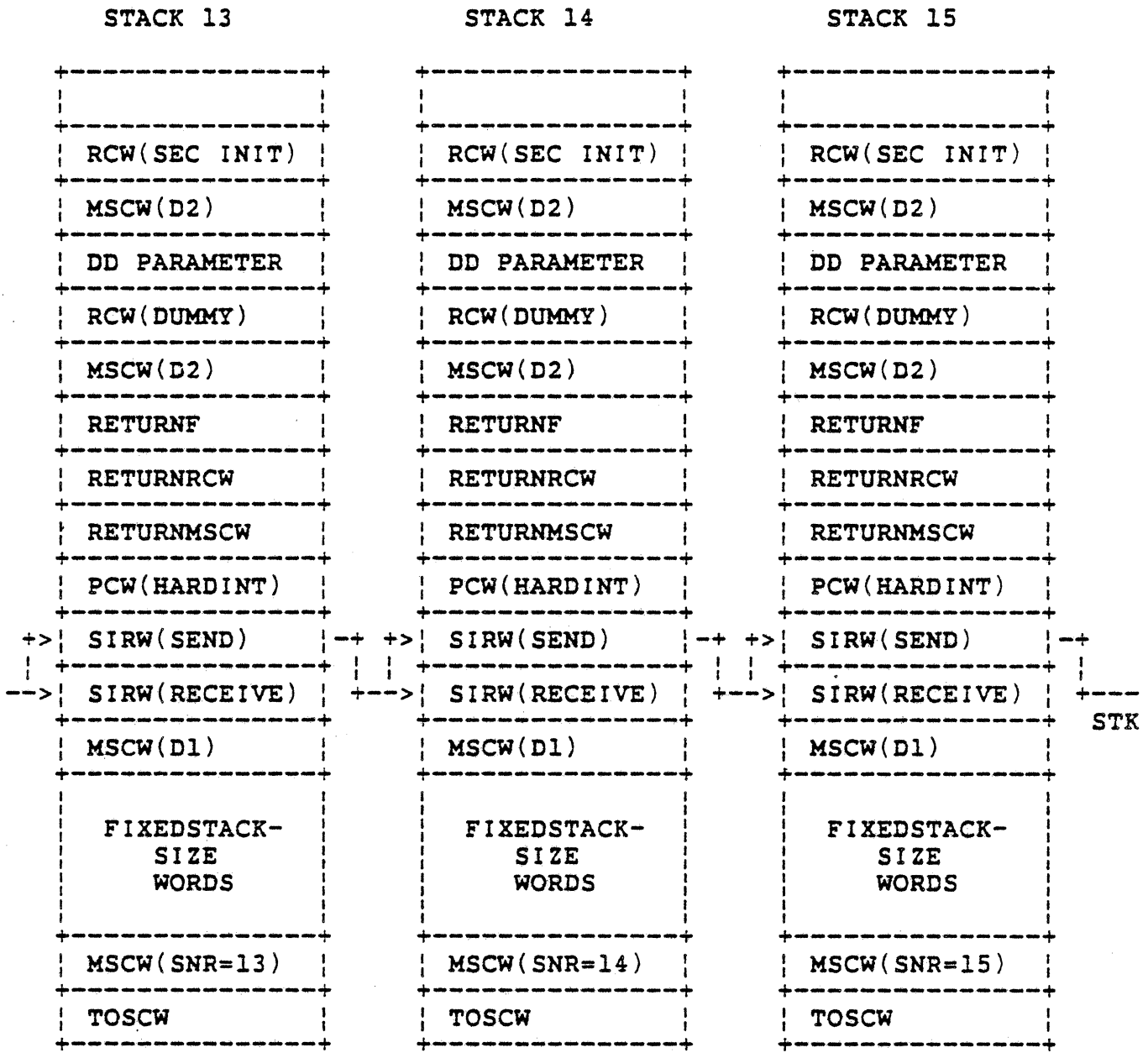
2. Exit into SECONDARYINITIALIZE.



STACKS ARE INITIALSTACKSIZE(150)+FIXEDSTACKSIZE WORDS.
THE HALT LOAD STACK HAS AN ADDITIONAL 1000 WORDS.

ALL STACKS HAVE THE TOSCW SET UP WITH F POINTING TO THE D2 MSCW AND S POINTING TO THE RCW FOR SECONDARYINITIALIZE.

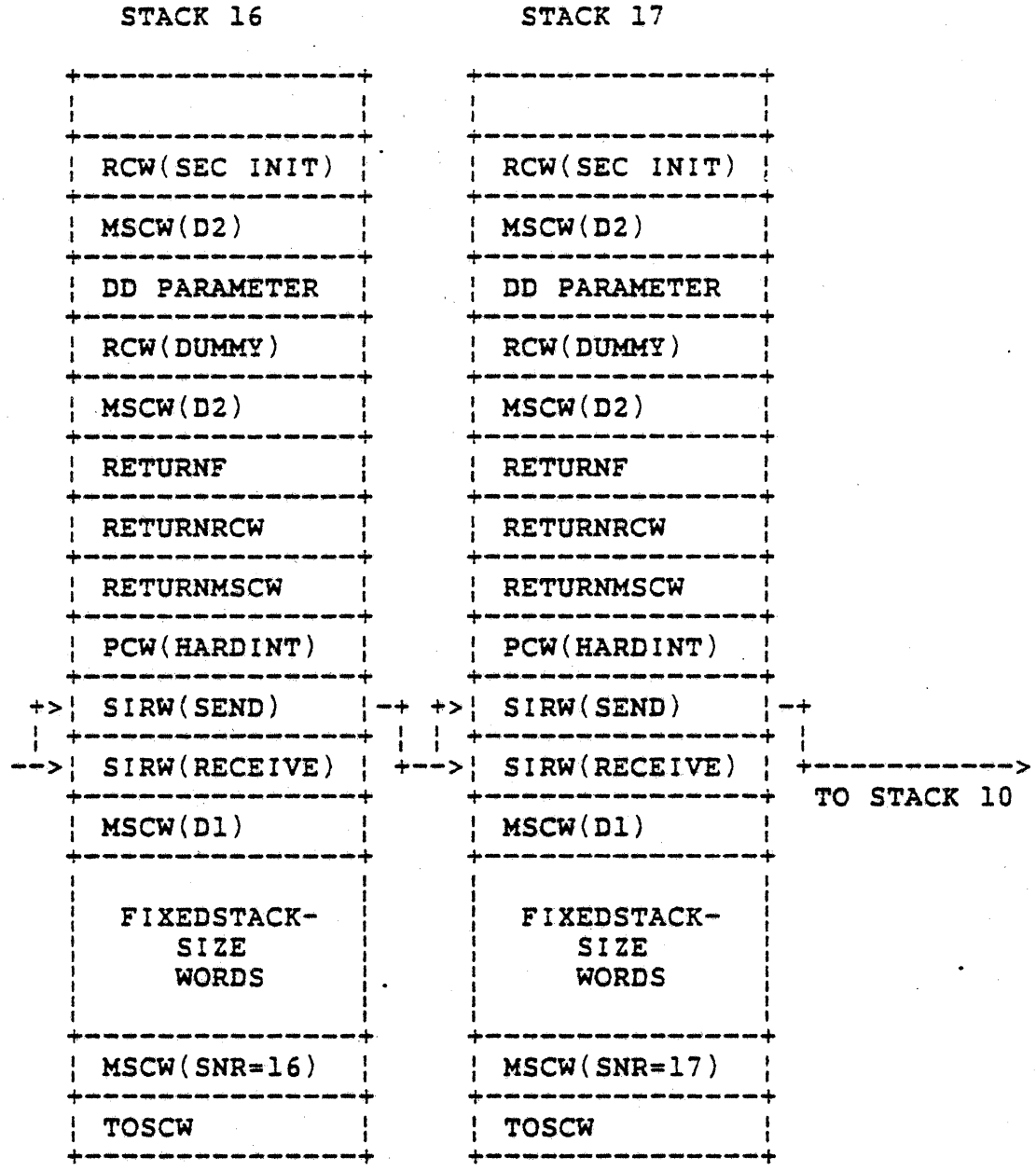
FIGURE 3-3. HALT LOAD STACKS (1 of 3)



STACKS ARE INITIALSTACKSIZE(150)+FIXEDSTACKSIZE WORDS.
THE HALT LOAD STACK HAS AN ADDITIONAL 1000 WORDS.

ALL STACKS HAVE THE TOSCW SET UP WITH F POINTING TO THE D2 MSCW AND S POINTING TO THE RCW FOR SECONDARYINITIALIZE.

FIGURE 3-3. HALT LOAD STACKS (2 of 3)



STACKS ARE INITIALSTACKSIZE(150)+FIXEDSTACKSIZE WORDS.
THE HALT LOAD STACK HAS AN ADDITIONAL 1000 WORDS.

ALL STACKS HAVE THE TOSCW SET UP WITH F POINTING TO THE D2 MSCW AND S POINTING TO THE RCW FOR SECONDARYINITIALIZE.

FIGURE 3-3. HALT LOAD STACKS (3 of 3)

SECONDARYINITIALIZE

SECONDARYINITIALIZE breaks the SIRW chain by writing a zero on the RECEIVEWORD in the current environment.

The following portion of the PRIMARYINITIALIZE outline is about SECONDARYINITIALIZE and is broken up into two parts. The first part is called SLAVEQUARTERS and is where the slave CPM's will stay during initialization. The Halt/Load CPM will never execute this code but the slaves won't execute any other code until the initialization process is completed. The second part of the section is simply referred to as the master's code and it is here that most of the work is done. The slaves while in their quarters will do only what they are told to do by the master processor. It is important to note here, however, that there is no master slave relationship among processors once the system is completely initialized.

OUTLINE OF SECONDARYINITIALIZE (Halt/Load CPM)

1. All processors are told to log on.

2. All DCP's are halted.

3. Global Memory is verified by each processor.

   A. Call SAVERETURN which will save current environment and return false.

   B. Try to test mod.

   C. If the mod is bad HARDINTERRUPT would have been entered. HARDINTERRUPT will use the environment saved above and return a true. The MOD will be marked offline.

4. Local memory is verified by the CPM that can see it.

5. Global memory is linked and MEMLOCK is initialized.

6. Local memory for each box is linked and MEMLOCK is initialized.

7. The BOXINFO array is established. Information generated earlier is moved to this array.

8. Some MCP arrays are set up in save memory.

9. Some tables generated earlier are moved to normal memory management areas.

10. The alternate D0 memory area is set up.

11. The size of the STACKVECTOR is computed. This is based on the number of memory mods on the system.

12. Arrays STACKINFO and STACKSTATUS are established.

13. The PIBVECTOR is established.

14. The STACKVECTOR is established.

15. Pseudo stacks are placed in the STACKVECTOR.

16. The MCP's PIB is set up.

17. GETAREA/FORGETAREA structures are established.

18. Special RCW's and PCW's are set up.

    A. Normal HARDWAREINTERRUPT routine is set up.

    B. SOPHIA is called to set up GEORGE and start idlers.

    C. BOJEOJ is called to set up NORMALBOJ and NORMALEOJ.

19. The ODT message value arrays are placed in a two dimensional array.

20. PERIPHERALINITIALIZE is called to set up the software and hardware peripheral environment.

21. The memory dump to tape PCW is put in the D0 stack.

22. FIBSTACK is called to set up the IOPCW's array used by logical I/O.

23. Independent runners AREAMANAGER, STARTSYSTEM, and CONTROLLER are forked.

24. ETERNALIR is initiated. One copy is initiated for each box. One copy is initiated for global.

25. The alternate stacks are forked.

26. All CPM's enter their copy of ETERNALIR or an idler.

ETERNALIR

ETERNALIR will finish initialization. ETERNALIR for the Halt/Load CPM will do the following:

1. Release initialization memory areas.

2. Fork ANABOLISM.

3. Call ANABOLISM to start the forked independent

runners.

ETERNALIR for a non-Halt/Load CPM will do no other initialization.

The following is a summary of ETERNALIR stacks running for each type of system.

1. Tightly coupled system:

   One ETERNALIR per box and one ETERNALIR in global.

2. Monolithic system:

   One ETERNALIR in the system.

At this point the system is interrupt driven.

SECTION 4

MCP COMPILATION AND BINDING

INTRODUCTION

The MCP consist of several different programs that must be compiled separately and then bound together to form a completely functional operating system. The following files are required to compile the sections of the MCP:

The compilers:

1. SYSTEM/NEWP
2. SYSTEM/DCALGOL
3. SYSTEM/BINDER

The input files:

1. SYMBOL/MCP
2. SYMBOL/ALGOLSUPPLEMENT
3. SYMBOL/CONTROLLER
4. SYMBOL/WFL
5. SYMBOL/JOBFORMATTER
6. SYMBOL/SCTABLEGEN

COMPILING THE MCP

The following card decks could be used to compile the sections of the MCP:

```
  COMPILE SYSTEM/MCPHOST WITH NEWP LIBRARY;

     COMPILER FILE CARD (TITLE = SYMBOL/MCPHOST);
```

```
DATA
$ MERGE
$ SET STACK LIST SINGLE XREF
?

COMPILE SYSTEM/ALGOLSUPPLEMENT WITH ALGOL LIBRARY;
   COMPILER FILE CARD (TITLE = SYMBOL/ALGOLSUPPLEMENT);
DATA
$ MERGE
$ SET STACK LIST SINGLE XREF
?

COMPILE SYSTEM/CONTROLLER WITH DCALGOL LIBRARY;
   COMPILER FILE CARD (TITLE = SYMBOL/CONTROLLER);
   COMPILER FILE MCPSYMBOLIC (TITLE = SYMBOL/MCP);
   COMPILER FILE SCTABLEGEN (TITLE = SYMBOL/SCTABLEGEN);
COMPILER DATA
$ MERGE
$ SET STACK LIST SINGLE XREF.
?

COMPILE SYSTEM/WFL WITH DCALGOL LIBRARY;
   COMPILER FILE CARD (TITLE = SYMBOL/WFL);
   COMPILER FILE MCPSYMBOLIC (TITLE = SYMBOL/MCP);
COMPILER DATA
$ MERGE
$ SET STACK LIST SINGLE XREF
```

```
?


COMPILE SYSTEM/JOBFORMATTER WITH DCALGOL LIBRARY;

   COMPILER FILE CARD (TITLE = SYMBOL/JOBFORMATTER);

COMPILER DATA

? MERGE

$ SET STACK LIST SINGLE XREF

?
```

BINDER

The program SYSTEM/BINDER is a compiler used to combine separately compiled programs, usually written in different languages, into a single BOUND program. This action is shown in figure 4-1. The HOST block is the program the SUBPROGRAM(S) must be bound to. The input files are not changed in any way but a new file will be created and is represented as BOUND PROGRAM in figure 4-1.

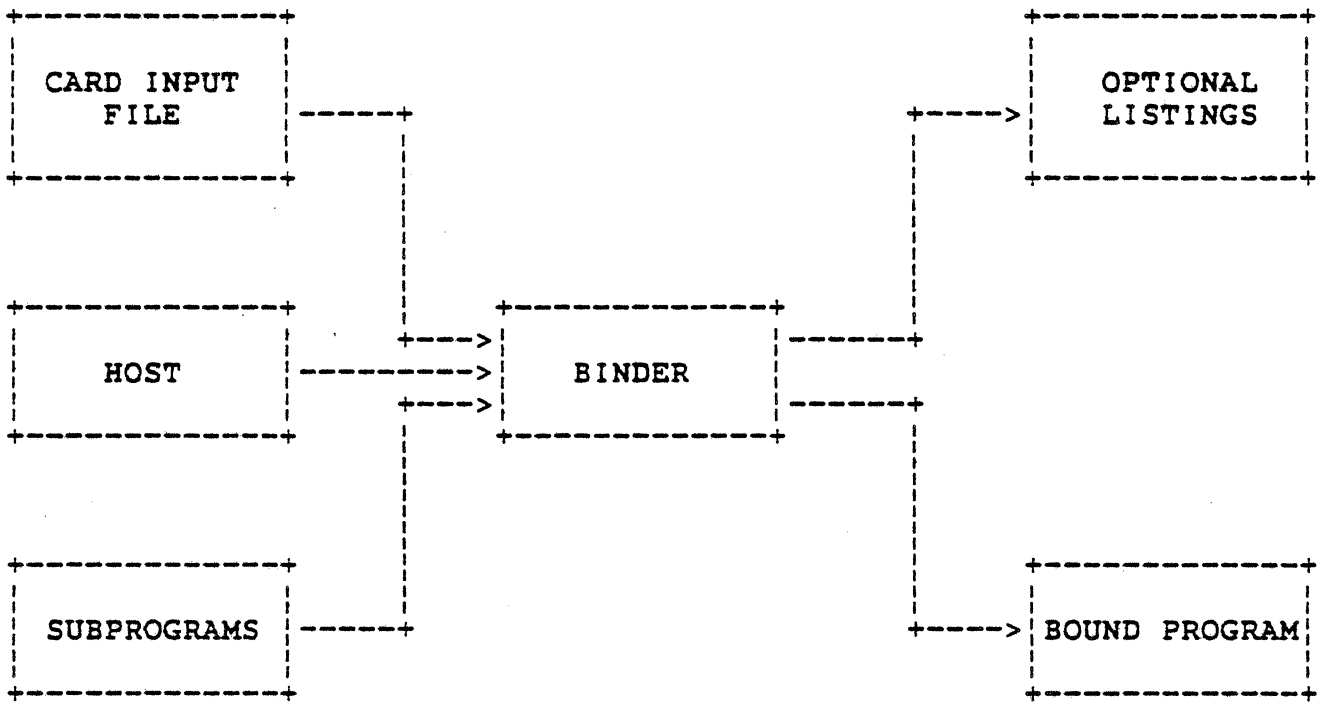


Figure 4-1. Binding Process

To produce an executable MCP the sections, previously mentioned, must be bound together. This may be done with the following deck:

```
BIND SYSTEM/MCP WITH BINDER LIBRARY;

BINDER DATA

  HOST IS SYSTEM/MCPHOST;

   BIND = FROM SYSTEM/ALGOLSUPPLEMENT, SYSTEM/WFL;

   BIND CONTROLLER FROM SYSTEM/CONTROLLER;

   BIND JOBFORMATTER FROM SYSTEM/JOBFORMATTER;

  ?
```

PATCH RELEASES

Patch releases are generated in the same way as a base release. When generating a patch release patches must be applied against the base release. In many cases a NEWTAPE must be generated because software items do includes on other software items. Thus, it is best to generate NEWTAPES before

any software is compiled. SYSTEM/PATCH can be used to generate the new symbolics.

SEPCOMP

The MCP can be SEPCOMP'ed. For information on this feature see the section on NEWP. A deck that could be used for SEPCOMP follows:

```
BEGIN JOB NEWP/SEP;
   TASK
     PT,
     CT;
   RUN SYSTEM/PATCH [PT];
      FILE TAPE(TITLE=SYMBOL/MCP);
      FILE PATCH(TITLE=PATCH/NEWPSEP);
   DATA
   $.COMPARE MARK
   $#
   $CLEAR LINEINFO MCP SEPCOMP "SYSTEM/MCP."
   <patches>
   ? % END PATCH INPUT
   IF PT(VALUE) NEQ 1 THEN ABORT "BAD PATCH";
   COMPILE SEP/MCP WITH NEWP[CT] LIBRARY;
      COMPILER FILE TAPE(TITLE=SYMBOL/MCP);
      COMPILER FILE CARD(KIND=DISK,
         TITLE=PATCH/NEWPSEP);
   IF CT ISNT COMPILEDOK THEN ABORT "BAD COPILE";
   REMOVE PATCH/NEWPSEP;
END JOB.
```

INTRINSICS BIND

A deck tht could be used to bind the intrinsics is shown below.

```
BIND SYSTEM/INTRINSICS BINDER LIBRARY;
BINDER DATA
$SET LIST INTRINSICS
BIND = FROM SYSTEM/ALGOLINTRINSICS,
            SYSTEM/DCALGOLINTRINSICS,
            SYSTEM/ESPOLINTRINSICS,
            SYSTEM/PLINTRINSICS;
BIND UDSTRUCTURETABLE FROM SYSTEM/UDSTRUCTURETABLE;
?
```

SECTION 5

INDEPENDENT RUNNERS AND SPECIAL STACKS

INTRODUCTION

This section is devoted to two very important topics: MCP Independent Runners (IR'S) and special MCP stacks set up during system initialization. There are many IR'S, some important, some not so important. The IR'S to be discussed in this section are those that are always present, namely, ANABOLISM, ETERNALIR, and IDLERSTACK. CONTROLLER (always present) along with another IR, CONTROLCARD (not always present), will be discussed later, in the section on process control. The special stacks discussed in this section are the DISKFILEHEADERS stack, INTRINSICS stack and the DATACOM QUEUE stack.

INDEPENDENT RUNNERS

Procedures in ALGOL may be RUN. Procedures in NEWP may be FORKED. The analogy is exact. When ALGOL procedures are RUN, the MCP creates a separate stack for the RUN procedure to execute in, asynchronously and independently. The same thing occurs when an MCP procedure is FORKED, that is, the MCP creates a separate stack for the procedure. These stacks are referred to as INDEPENDENT RUNNERS and exist for some very good reasons. Perhaps the most obvious reason of which is to get the MCP off of user stacks when the MCP code to be executed may have to wait on events or special system locks or operator responses from RSVP messages.

The following paragraphs discuss how IR's are forked and what some of the more important IR's do.

The FORK Statement

The fork statement provides a method for initiating independent runners. The procedure referenced may be typed or untyped.

The NEWP compiler assumes there is a procedure at a fixed location in the MCP's stack that will handle the procedure's initiation. This procedure is named FORKHANDLER. A typical example of a fork statement has been taken from the MCP.

```
FORK ETERNALIR [GLOBALBOX,IRSTACKSIZEB,IRPRIORE]

GLOBALBOX is defined as 1

IRSTACKSIZEB is defined as 380.

IRPRIORE is defined as 101.
```

INSTRUCTION	COMMENT
MKST	Start of the fork statement's code.
NAMC 0,24C	ETERNALIR's PCW.
	If the procedure being forked was to be passed parameters, they would be located here.
MKST	Prepare to enter FORKHANDLER.
NAMC 0,AF	FORKHANDLER's PCW.
ONE	Box number.
LT16 17C	Size of the process stack required for this IR.
LT8 65	Priority of this independent runner.
LT48 09C5E3C5D9D5	This instruction and the following two instructions
LT 48 C1D3C9D90000	pass the name of the IR to FORKHANDLER.
JOIN	Make the two name words double precision.
ENTR	Enter the FORKHANDLER procedure.
ENTR	This enter would appear to be on ETERNALIR's PCW but what will actually happen is an enter on JUSTEXIT will occur. When FORKHANDLER is executed, the IRW pointing to ETERNALIR's PCW will be replaced with an IRW pointing to JUSTEXIT's PCW. JUSTEXIT will simply exit back into the procedure that invoked the fork to begin with.

FORKHANDLER Procedure

FORKHANDLER is the procedure called when the fork statement is executed. This procedure builds a fork queue entry and then inserts this entry into the fork queue. After an entry has been placed in this queue ANABOLEVENT is caused and from this point on, it is up to ANABOLISM to initiate the independent runner. Causing ANABOLEVENT has the effect of placing ANABOLISM in the READYQ. It is important to note here that ANABOLISM itself is an independent runner and must be started in a different manner. How could ANABOLISM start itself up if it didn't exist to begin with?

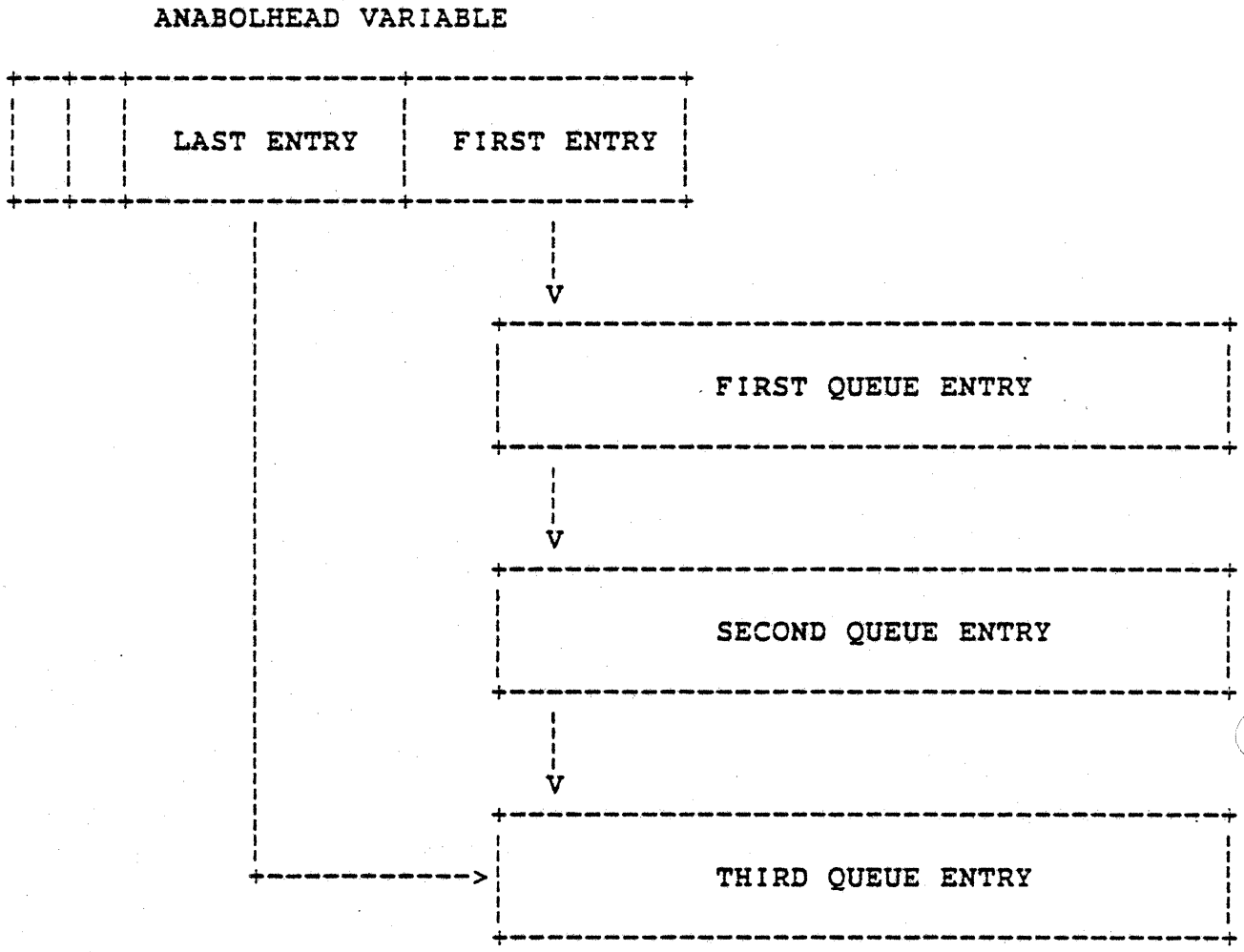


Figure 5-1. FORK Queue

ANABOLISM Routine

The purpose of ANABOLISM is to start up independent runners. This procedure performs the same basic function as DOCTOR does for regular tasks. ANABOLISM works on a queue of requests initiating one entry at a time until the queue is empty. When the fork queue is empty, ANABOLISM goes to sleep on ANABOLEVENT.

When ANABOLISM is awakened by ANABOLEVENT having been caused, it looks in the queue and delinks the first entry, acts on it and then continues until the queue is empty at which time

ANABOLISM goes back to sleep on ANABOLVENT.

FORKHANDLER is the procedure that inserts requests into the queue. This queue works in such a way that ANABOLISM locks FROCK only while it is delinking requests from the queue. Thus, FORKHANDLER can insert entries into the queue while ANABOLISM is acting on them.

Figure 5-1 shows a global variable, ANABOLHEAD, that is used to point to the queue of requests. GETAREA and FORGETAREA are used for the request spaces.

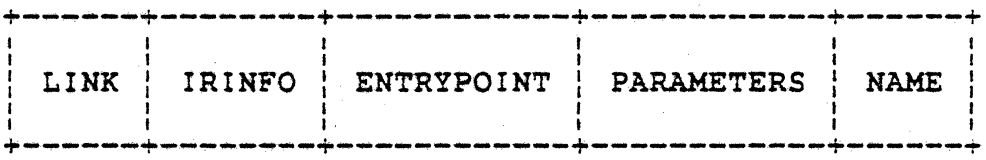


Figure 5-2. Queue Entry

A field in ANABOLHEAD contains the absolute address of the first entry in the queue and another field in ANABOLHEAD contains the absolute address of the last entry in the queue. The first word of each entry contains a link to the next entry. The entries are variable length and may be seen in Figure 5-2.

ANABOLISM'S basic function when acting on an entry is to get space for a PIB and transfer information from the queue entry to this array. The information transferred is the stack's priority, size, entrypoint, and name. If there are parameters, they are left in the area. A word in the PIB will point to the area. If there are no parameters, the area is released by calling FORGETAREA. INITIATE is called passing the PIB as a parameter. INITIATE, as always, gets space for and initializes a process stack and, depending on memory availability, etc. will place the stack in the READYQ or the SHEETQ.

ETERNALIR Routine

ETERNALIR is forked by SECONDARYINITIALIZE during system initialization and can act as five different independent runners. Its purpose is to always be around when one of these jobs is to be performed, thus not requiring five separate independent runners to be in memory for these relatively small tasks.

The five jobs that ETERNALIR does are:

1. Force a scheduled task to run.

2. Initiate a task in the box the task is to run in. Used in tightly coupled task initiation.

3. Terminate a task. Release the process stack.

4. Call MESSER to put out an ODT message.

5. Save failure analysis counters.

ETERNALIR also does the following:

ETERNALIR in Global box:

Get tasks out of the schedule and into the active mix.

Check tasks that have a WAITLIMIT specified.

Check peripheral status.

All ETERNALIR:

Poll MCM fail registers.

When one of the other MCP procedures wants ETERNALIR to act like one of the five IR's it calls RILANRETE passing three parameters which are subsequently linked into a list. RILANRETE then causes ETERNALEVENT to wake up the proper ETERNALIR.

Outline of ETERNALIR

1. Finish MCP initialization.

2. Every second the ETERNALIR in the Global box will:

   Check on tasks in the schedule queue.

   Check on tasks with a WAITLIMIT.

   Check peripheral status.

3. Every second all ETERNALIR's will check the MCM fail registers.

4. Wait for one second or the event for this ETERNALIR to be caused.

5. If the event was caused, the queue driven function as outlined above is done.

6. Go to step 2.

RILANRETE Procedure

RILANRETE is called when an entry is to be made in an ETERNALIR queue. This procedure is passed three parameters two of which will be placed in a two word area obtained by GETAREA. RILANRETE will insert the two parameters passed to it into the area and then call QINSERT passing the location of the area and the head/tail word of the queue as parameters.

QINSERT will link the area into the queue for the ETERNALIR specified by the head/tail word of the queue. It will then CAUSE the proper ETERNALIR event.

IDLERSTACK Procedure

IDLERSTACK is the procedure forked by SECONDARYINITIALIZE during system initialization that is used as a place for processors to idle while looping through GEORGE's code after GEORGE has found nothing else to do.

There is an 8 word array called IDLERSTACKS that is associated with this procedure and is indexed by processor box number. As each processor enters IDLERSTACK for the first time this array will have the stack number placed in it in the location specified by the box number, thus allowing the same processor using this array's information to always return to the same IDLERSTACK.

SPECIAL STACKS

The special stacks to be discussed here are the DISKFILEHEADERS stack, INTRINSICS stack and the DATA COMM QUEUE stack. the INTRINSICS stack, set up by the procedure INITIALIZEINTRINSICSTUF may not be set up at system initialization time as are the other two but it may be set up later as a response to the CI ODT message.

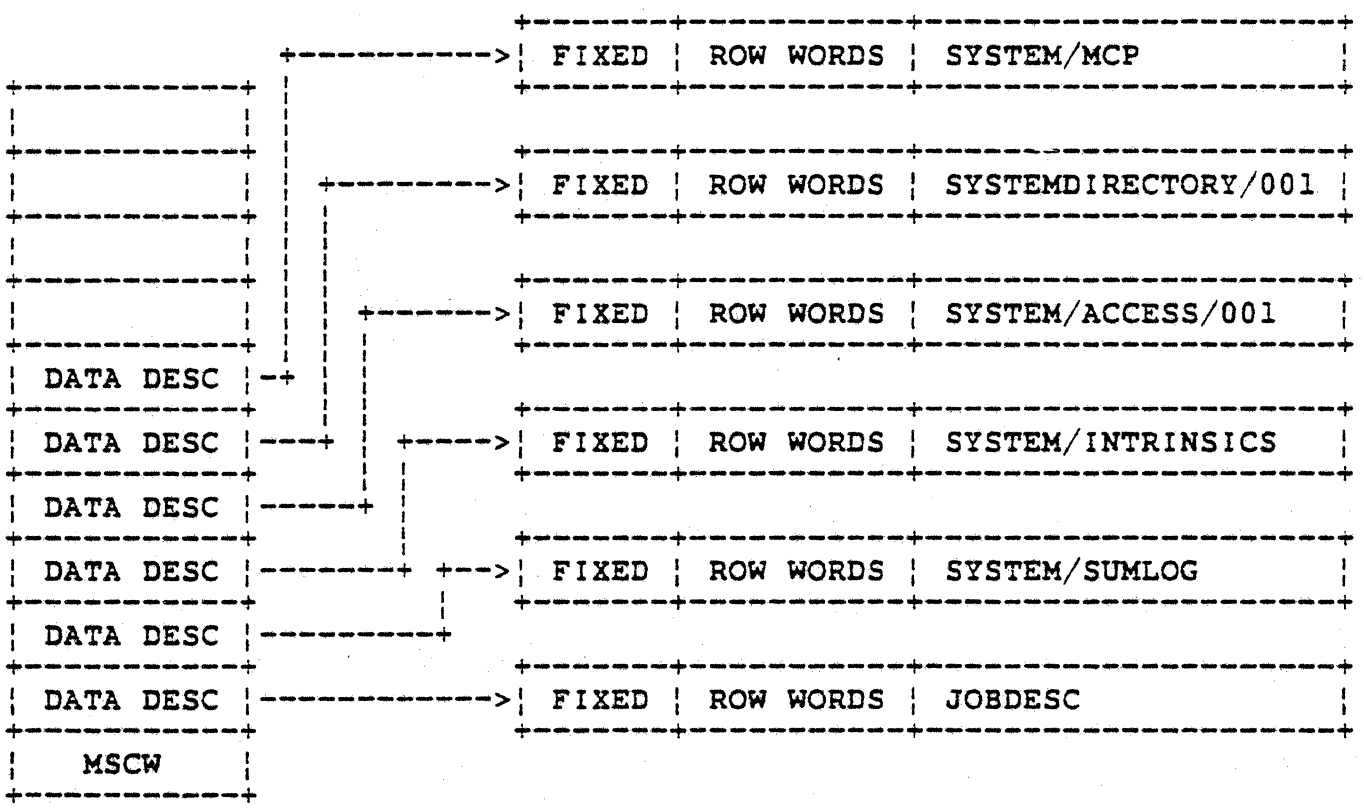
The DISKFILEHEADERS Stack

HEADERS are arrays normally saved on disk in the FLAT DIRECTORY and contain general information about the disk file they are associated with, as well as ROW ADDRESS WORDS that contain the absolute disk addresses of the rows of the file. When a disk file is opened, the file's header is read into memory (if the file existed before, of course). While in memory, information from the header (such as EOF, number of rows, etc.) may be obtained. When a header is read into memory, a descriptor pointing to it is placed in the DISKFILEHEADER stack. This linkage remains until the file is closed, at which time, the header is updated (i.e, timestamped) and then written back into the FLAT DIRECTORY.

The headers that are nearly always seen in the header stack, as well as the stack itself, are shown in Figure 5-3. Note that the SYSTEMDIRECTORY header is for the FLAT DIRECTORY and that the JOBDESC header is used by CONTROLLER to save the rules for controlling ODT screen activity, job queue information, etc. The JOBDESC file also contains the JOB QUEUES, a linked list of headers that point to the JOBFILES created by the WFL compiler.

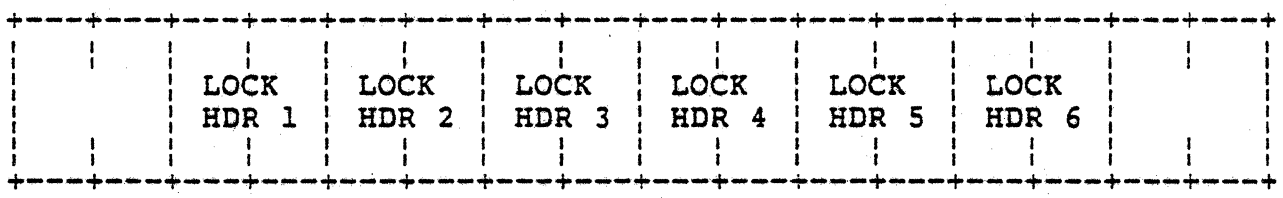
The first word in the DISKFILEHEADERS stack is an MSCW. This allows the data descriptors in the stack to be referenced with SIRW'S.

DISK FILE HEADERS STACK



Each DISK FILE HEADER has a fixed part, a variable number of ROW ADDRESS WORDS and a variable length NAME. The HEADER is pointed to by a DATA DESCRIPTOR in the DISK FILE HEADERS STACK.

HEADERLOCKS



Each HEADER in the DISK FILE HEADERS STACK has a LOCK entry associated with it. This LOCK is contained in a EVENT ARRAY called HEADERLOCKS. The DISK FILE HEADERS STACK and HEADERLOCKS are the same length.

Figure 5-3. Disk File Headers Stack

The INTRINSICS Stack

The INTRINSICS stack is the stack, created by the MCP procedure INITIALIZEINTRINSICSTUFF from the segment dictionary of an INTRINSICS file. The INTRINSICS stack contains PCW'S to intrinsics and some other miscellaneous information.

Intrinsics, as used in this text, are any procedures whose PCW'S are in the INTRINSICS stack, e.g., SIN, COS, SORT. This eliminates such procedures as PROGRAMDUMP or BLOCKEXIT although these procedures may rightfully be called intrinsics.

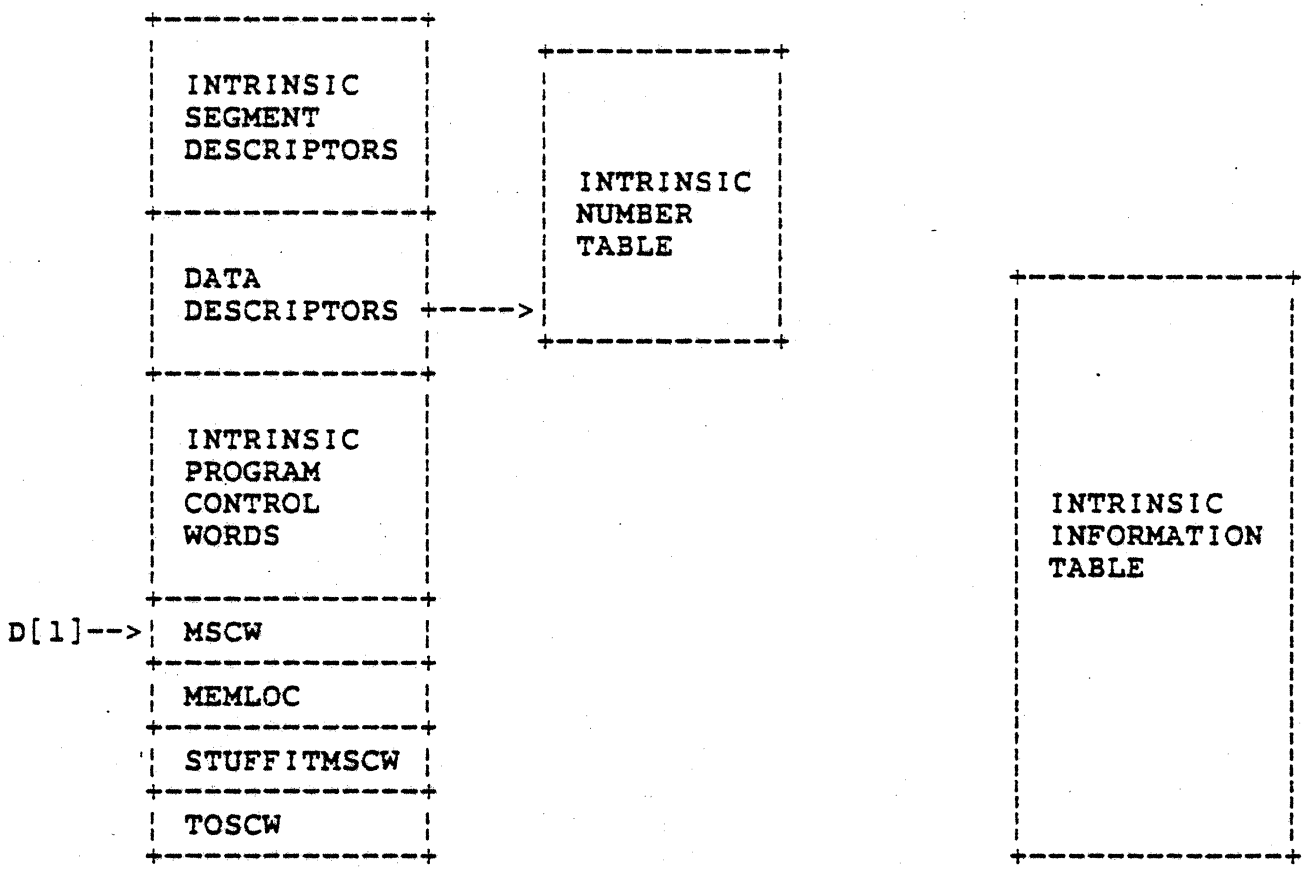
An intrinsic's PCW is located by an identifying INSTALLATION number and INTRINSICS number. These two numbers are found by compilers by use of the INTRINSICS INFORMATION TABLE shown in Figure 5-4. To find an INTRINSIC, the MCP must be supplied with a word containing the INSTALLATION number in bits 23:11 and the INTRINSICS number in bits 12:13. Given this word, the MCP will use it as the argument of a MASKSEARCH through the INTRINSICS NUMBER TABLE (This table is pointed to by the INTNUMF of the STUFFITMSCW in the INTRINSICS stack). The index returned by the MASKSEARCH instruction is used as an index, D[1] relative, that points to the intrinsic's PCW in the intrinsics stack.

For each INTRINSIC referenced in an object program, a segment dictionary location in that program's segment dictionary will be assigned. This location will have a tag of 5 and bits 42:3 will equal 7. The remainder of this word will be as described in the paragraph above.

At execution time, the first reference to an INTRINSIC will result in an INVALID OPERAND interrupt due to attempting to enter a tag 5 word instead of a PCW. At this point, the MCP will check to see if the interrupt was on an ENTR instruction and if it was, a check is made to see if the word causing the interrupt is a valid intrinsic word as described above. If all checks are passed okay, the MCP zeroes out the tag field and bits 42:3 and then uses the new word with the MASKSEARCH instruction on the INTRINSICS NUMBER TABLE. Having obtained an index from the MASKSEARCH operator, the MCP builds an SIRW that points to the appropriate PCW in the INTRINSICS stack and then replaces the data descriptor in the object program's segment dictionary with the newly created SIRW. Next, the interrupted operator is re-executed. Note that when an INTRINSIC actually is entered, DISPLAY REGISTER 1 will point at the MSCW in the INTRINSICS stack (due to entering via an SIRW) and DISPLAY REGISTER 2 will point at the appropriate MSCW in the object program's process stack.

It should be mentioned here that the MCP locates the INTRINSIC INFORMATION TABLE in the INTRINSIC's file by use of word 17 (decimal) in SEG 0 of the INTRINSIC's file. This word is known as SOINTRINSICTABLE. The INTRINSIC NUMBERS TABLE is located by use of word 1 (decimal) in SEG 0. This word is

known as SOINTRINSICNUMBERS in the MCP.



STUFFITMSCW.INTNUMF points to DATA DESCRIPTOR that points to INTRINSIC NUMBER TABLE.

INTRINSIC INFORMATION TABLE contains a description of each INTRINSIC including parameter information and INTRINSIC number. This TABLE is used by compilers to check and generate calls. It is pointed to by INTRINSICINFO.

Figure 5-4. Intrinsics Stack and Associated Arrays

DATA COMM QUEUE Stack

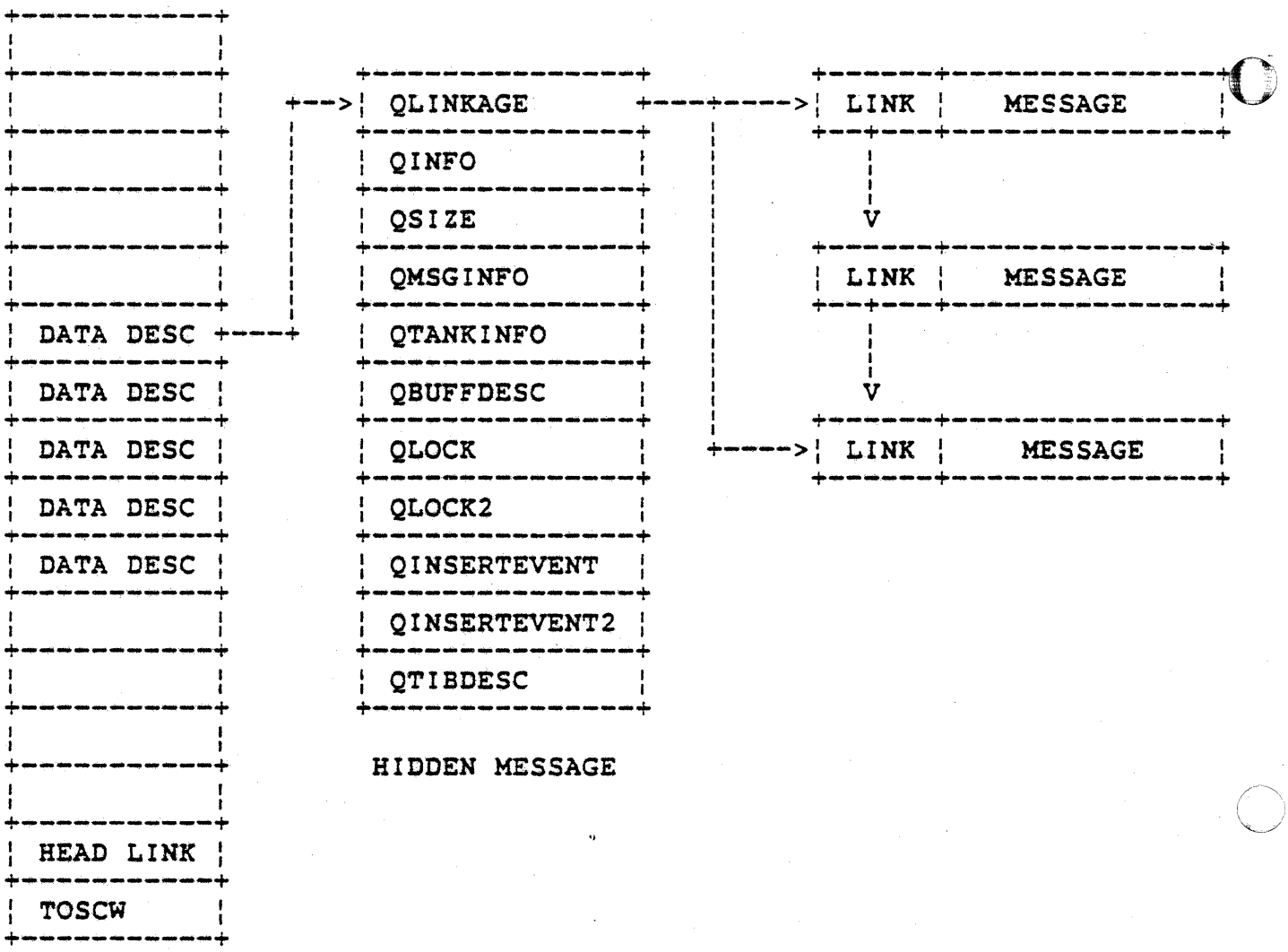
Besides being a central location for queues declared in DCALGOL and FILE INPUT QUEUES for programs with remote files, the DATA COMM QUEUE stack serves as the communication link among Independent Runners. The main IR'S using the queues in this stack are CONTROLLER and CONTROLCARD (WFL).

The DATA COMM QUEUE stack can be seen in Figure 5-5 The first word in the stack contains a TOSCW and the second word contains a link.

This link is the start of a linked list of available queue locations. This list consists of numbers relative to word one, itself. Locations in this stack that are not a part of this list are data descriptors that point to HIDDEN MESSAGES. These HIDDEN MESSAGES contain head and tail pointers of queue entries plus attributes of the queue such as memory limit, population, etc.

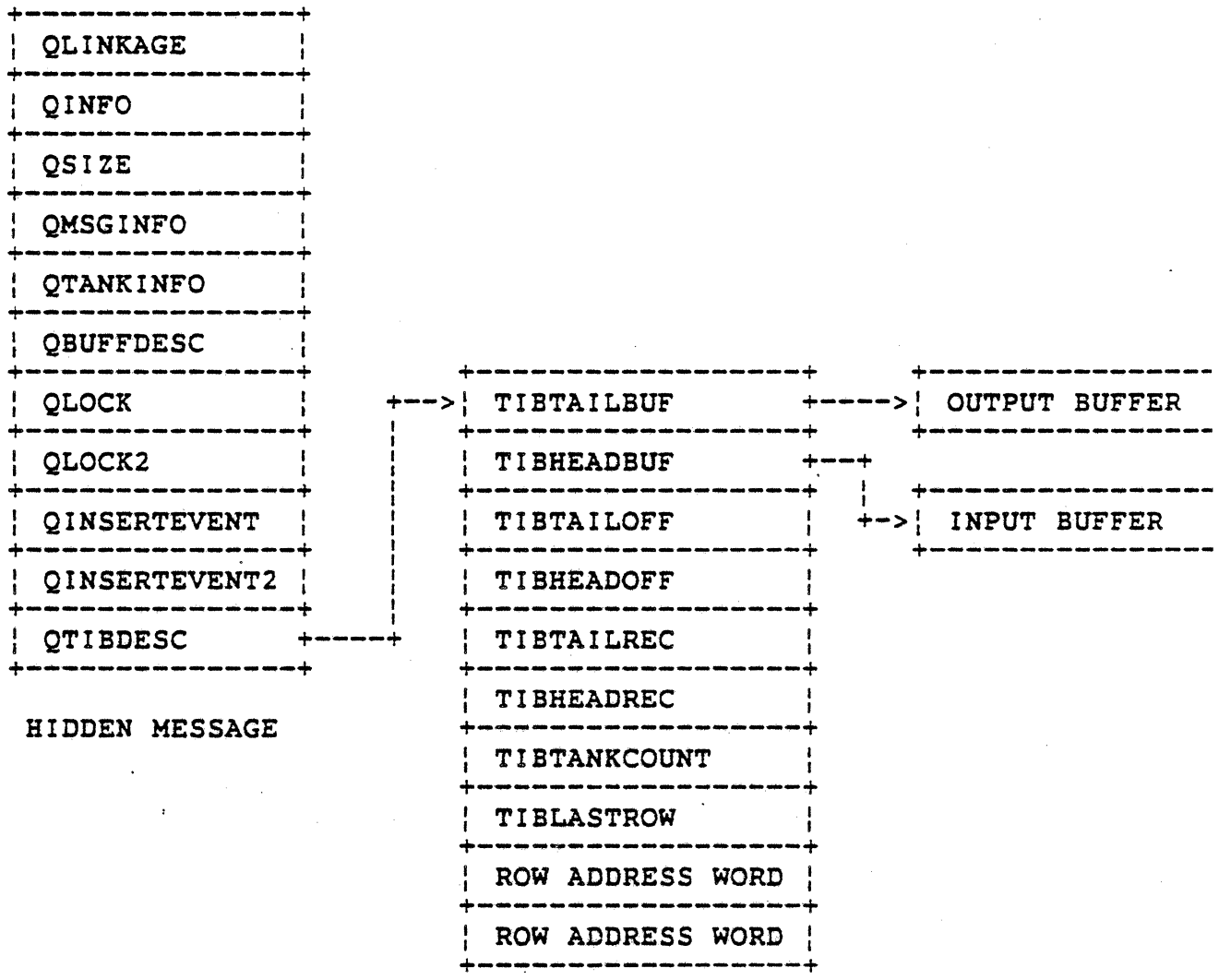
As messages are placed in a queue, as can be seen in figure 5-5, the queue's memory limit may be reached. If the queue's limit is reached, a TANK INFORMATION BLOCK (TIB) is built, disk rows are obtained with the procedures GETUSERDISK and subsequent messages for the queue are TANKED, (i.e., written to disk). It should be pointed out that queue messages will not be found in central memory and disk at the same time; they will either be queued on disk or in memory, not both.

Figure 5-6 shows more detail on the HIDDEN MESSAGE and TIB.



Word one in the stack is the beginning of a linked list of available stack locations. Other locations point to the HIDDEN MESSAGE.

Figure 5-5. Queues



The TIB contains a variable number of ROW ADDRESS WORDS. These ROW ADDRESS WORDS point to the disk area.

Figure 5-6. Hidden Message and Tank Information Block

SECTION 6

HARDWAREINTERRUPT PROCEDURE

INTRODUCTION

The procedure HARDWAREINTERRUPT77 provides the interface between the B 7000 hardware and software. This procedure is called by both, the hardware and software. The hardware calls this procedure to allow the MCP to take care of presence bit interrupts, open files, establish queues, handle hardware failures, etc.

The section of the B 7000 System Reference Manual on interrupts should be read for information regarding hardware actions during the various types of interrupts. For software actions, see the outline on the HARDWAREINTERRUPT77 procedure.

Outline of HARDWAREINTERRUPT77

1. If the processor id. is not equal to the stack number, the processor is not on a control mode 2 stack. A branch is made to one of the following labels:

   EXTRNL

   INTTIMER

   PBIT

   ALLOTHERS

2. If the processor is in a control mode 2 stack, a branch is made to CM2INTERRUPTS.

3. At a label called away HARDWAREINTERRUPT77 will leave. If the stack was not executing GEORGE(LOAFING) it will call GEORGE otherwise it will exit back to GEORGE. The LOAFing bit will prevent recursive calls on GEORGE.

4. At a label called PBIT, the present bit interrupt is handled. Procedure PRESENCEBIT is called to make the area present. HARDWAREINTERRUPT77 exits back to the

user.

5. At a label called EXTRNL external interrupts are handled. The interrupts handled here are:

   A. Completed I/O's and status change.

   B. IOM errors.

   C. Processor to processor interrupts.

6. At a label called INTTIMER HARDWAREINTERRUPT77 will handle an interval timer interrupt. It will exit back to GEORGE or call GEORGE.

7. At a label called ALLOTHERS, the other interrupts are handled. These interrupts are:

   A. Segmented array. Call SEGARRAYINTERRUPT.

   B. Stack overflow. If the stack is processing a stack overflow, a memory dump is taken; otherwise, the limit of stack is moved into the overflow area. KANGAROO is called to stretch the stack.

   C. Invalid Operator. HARDWAREINTERRUPT77 will decide if the interrupt is a true error or the user is:

      a. Linking to a library.

      b. Making an intrinsic reference for the first time.

      c. Calling a procedure that is not bound.

      d. Opening a queue.

      e. Opening a file.

      These Invalid Ops are set up by the compiler. HARDWAREINTERRUPT77 will call the proper procedure to perform the function requested.

   D. Exponent underflow. The word is set to zero.

   E. Invalid index.

   F. Memory fail 1. Call MEMORYERROR.

   G. Memory parity. Call MEMORYERROR.

   H. Processor internal. Log the error and retry the instruction. If it fails the retry the stack is DS-ed.

I. Memory parity or memory fail 1. Call MEMORYERROR.

J. Egg timer. Take memory dump.

K. Invalid address. Count as an error against the processor.

L. Loop timer. Handle as processor internal interrupt unless the operator was a link list lookup.

8. Search for on-fault statement. If there is an on-fault statement it is executed. If the program has not handled the interrupt and a return to the user was not done above, the stack is DS-ed. If the stack is an independent runner a dump is taken.

9. At a label CM2INTERRUPTS, the following is done:

A. Log the rror.

B. If the old stack was not an independent runner and the interrupt was a loop timer or processor internal, the CPM will move back to the old stack and DS it.

SECTION 7

MEMORY MANAGEMENT

INTRODUCTION

This chapter provides information regarding the implementation rationale and use of the virtual memory management technique utilized on the Burroughs large systems. Before proceeding, it is necessary to define a few terms and certain aspects of the memory management techniques.

VIRTUAL MEMORY

The memory management scheme is based on the virtual memory concept. Essentially, the language compilers, when operating on a program, divide the program's code into a number of segments. Associated with these segments are segment descriptors. These segments and a dictionary of segment descriptors are kept on some memory extension device such as head-per-track disk. Additionally, data arrays are also segmented and a data descriptor is associated with each data array. The descriptor contains fields which indicate the segment length, the base address of the segment, and a flag (or presence bit) which denotes whether the base address is a main memory address or a disk address.

The operating system maintains a linked list of available main memory space segments. When a program, during execution, attempts to address data or code which currently resides only on disk (presence bit off), a hardware interrupt is generated. The hardware, on sensing this interrupt, will automatically call the HARDWAREINTERRUPT procedure passing the descriptor as a parameter. This procedure will in turn, call procedures that get space in central memory, adjust the descriptor to point at this space, and load the segments from disk to this space. If the data space has not been previously referenced, the disk address of the descriptor would have been zero. The presence bit routines detect this and bypass the load of disk to main memory.

OVERLAY Mechanism

When the presence bit routines are attempting to load a segment from disk, a possibility exists that they cannot find an area in the available space list large enough to hold the segment. When this occurs, some of the in-use segments must be overlayed. Each in-use segment is preceded by a memory link which contains the address of the descriptor pointing at this segment and the home disk address of the segment. The system also contains a left-off pointer (that is, the address of some segment in main memory where the last invocation of the overlay process left off).

The overlay process consists of examining one or more consecutive segments, starting with the left-off pointer, fixing the descriptor pointing at this segment by turning its presence bit off and changing its address field from a memory to disk address. For code segments, this completes the process of overlaying a segment. Since code is not modifiable, the code segment on disk is always identical to code segment in main memory and, therefore, need not be returned to disk. For data segments, however, it is necessary to write the segment back to disk. The system maintains an overlay disk file for each program. The data segment is written to disk with the disk address being recorded in the data descriptor.

This overlay mechanism proceeds until enough consecutive space has been overlayed to satisfy the original demand. On completion of an overlay, the left-off pointer is set to point at the segment beyond the last one overlayed.

One of the undesirable effects of a virtual memory system based on the variable length segment approach is the checkerboarding effect. After the system has been running for a while, memory tends to contain a large number of small available areas, too small to be usable. This overlay mechanism utilizing the left-off pointer tends to reduce this problem as these small areas are eliminated during the overlay process.

Thrashing

A phenomenon known as thrashing can occur in any type of virtual memory system. The situation can occur where a high percentage of demands for a new segments invokes the overlay mechanism. The systems spends so much time overlaying that performance is degraded beyond tolerable limits. The situation occurs because the system has accepted for execution more programs than can conveniently fit in main memory.

Two techniques can be used to resolve the thrashing problem. One technique is to have the compilers analyze the program

being compiled and generate an estimate of the program's memory requirement. The system will accept a program for execution only when available main memory exceeds this memory estimate. Due to the dynamic nature of some languages, the compile-generated estimate can be extremely inaccurate. More elaborate techniques to generate accurate estimates have been considered but none have been found that are economically feasible. A second technique is to have the system detect when thrashing occurs and suspend execution, on a priority basis, of one or more programs. The overlay mechanism will roll out the suspended program's data and code segments, thus reducing thrashing. Most attempts to do this involve monitoring a certain limit. The problem with this technique is that an overlay rate which represents thrashing for one application or configuration does not necessarily represent thrashing on some other system.

Overlay Control

The discussion to this point was concerned with certain terminology and in general terms, the memory management scheme used on the large systems. Studies of this virtual memory management scheme have included plotting a program's memory utilization as a function of its overlay rates. The results for most programs are indicated in figure 7-1. The following observations can be made regarding these results:

A. The amount of core a program is using is dependent on its overlay rate. Most installations have an accounting mechanism to change for resources utilized by a program. One of those resources charged for is core utilization. The overlay mechanism previously described operates on a demand basis. Thus, the amount of overlay and therefore, the memory being utilized by a program varies widely, dependent on what other programs are being multi-processed with it.

B. The overlay rate has a large influence on the behavior of a program, thus, allowing a user control of the overlay rate would allow a degree of control of the program.

C. A constant but continuous overlay rate tends to keep core free of unused segments. The value the system keeps for the amount of main memory available becomes more meaningful. As a result, the system can provide better automatic scheduling capabilities. This also in turn tends to decrease the thrashing problem.

D. The curve of figure 7-1 shows a point where there is an abrupt change in slope. Mr. Denning [1] [2] defines the term working set as, that minimum collection of a program's segments which must reside in main memory for a process to run efficiently. That point on the curve in figure 7-1 where the slope o the curve changes abruptly

represents a program's working set. This point also indicates that there is an optimum overlay rate for a program. An overlay rate higher than this point represents degraded program performance, while overlay rates below this optimum represent inefficient memory utilization.

As a result of the stated observations, an overlay control capability was added to the system. This essentially consists of allowing the user to specify an overlay goal and then, on the basis of this overlay goal and program priority, generate and maintain a continuous and constant overlay rate against every program being multi-processed.

Implementation

The Overlay Goal

Two operator messages, SF and OG, can be utilized to set the overlay goal. The SF allows setting a system default overlay goal. The units of this overlay goal are in terms of percentage of overlayable core per minute. This default rate is used in conjunction with a program's priority to generate a program overlay rate as follows:

```
OLAYRATE (SN) = INTEGER(((100-PIB[PRIORITY])/50)*OLAYGOAL)
```

Essentially, the above rate sets priority 50 programs to an overlay rate equal to the system's default rate. Priority 99 programs will have an overlay rate of zero, while priority 01 programs will have an overlay rate of twice the system's default rate. The default overlay goal is also the overlay rate for the code and data segments of of the MCP. The OG message allows setting an overlay rate for a particular program, thus overriding the system generated default rate.

The implementation consists of an overlay control procedure (called WSSHERRIFF) which cyclically wakes ups at approximately 3 second intervals. This procedure first generates an overlay amount for each program in the system (including the operating system) by multiplying each program's current core utilization by the program's overlay rate. This procedure will then, starting with the left off pointer (described previously), examine consecutive segments of memory, moving the left off pointer to the next adjacent segment. If the owner's overlay amount is not zero, it is reduced by the size of the segment, and the segment is overlayed. This operation proceeds until the overlay amount of all programs in the system is reduced to zero. This process of imposing a fixed amount of overlay on every program in the system, as well as enhancing program predictability, can also enhance throughout, in that unused segments tend to

get overlayed, thus making room for more programs. This is done at the expense of elapsed time, i.e., the enforced overlay does slow down an individual program somewhat (the amount being dependent on the overlay rate). There does exist, however, certain high priority programs where the criteria is to have these programs execute as quickly as possible. For this reason, the option exists to turn off the overlay control procedure at any time by setting the system overlay goal to zero. This will automatically cause the overlay control procedure to deallocate any memory resources it is utilizing and terminate itself (which causes the system to deallocate its code and stack space). Setting the overlay goal to non-zero will automatically reinstate the overlay control procedure. Turning the overlay control procedure on and off can be done dynamically at any time, even when there are programs running.

On a tightly-coupled system there is one copy of WSSHERRIFF per box. Each copy will function as stated above. All copies of WSSHERRIFF are invoked when the overlay goal is non-zero.

Computing the Working Set

It was previously pointed out that it is not always possible for compilers to generate accurate core estimates for programs. It was also pointed out that by imposing a constant overlay amount on a program that at any instant of time, the memory being utilized by a program is its working set. It is also apparent that while a program's working set will not change significantly during small increments of time, a program's working set may undergo a large change over a long period of time. An effective working set size is of much more interest to both the user and the system than the instantaneous working set size. For the user, it provides a more accurate cost accounting. For the system, it provides enhanced automatic scheduling capability. The effective working set can be defined as the integral of the instantaneous working set over the program's running time. For each program, whenever its memory utilization is changed, the system generates the following:

A. CORINUSE := COREINUSE + AMOUNT

B. COREINTEGRAL := *+(AMOUNT * (PROCTIME + IOTIME))

where AMOUNT is the number of words (the sign of AMOUNT is plus if the segment is being added and minus if the segment is being overlayed). The effective working set of a program at any time (including end-of-job time) is then:

WSSIZE := COREINTEGRAL + (COREINUSE * (PROCTIME + IOTIME))

The value of the current WSSIZE is displayed on the console. (See the CU message.) At the end-of-job time, WSSIZE is

written out to the system log (where it appears as COREUSAGE) and also is used to update a program's core estimate if that program resides in the system directory. Additionally, WSSIZE is used in the control of the automatic program suspension/ resumption mechanism described next. The effective working set size WSSIZE is computed at all times whether or not the working set memory control is running.

Automatic Program Suspension/Resumption

The term thrashing was previously defined. Thrashing can now be redefined as a degraded state of system behavior due to the sum of all programs current working set exceeding the system's main memory capacity. One of the reasons for thrashing, specified earlier, was the inability of compilers to produce an accurate core estimate for a program. This, in turn, causes the system to accept programs for execution which will not fit in memory.

Another reason for thrashing is that some program's working sets can change significantly over a period of time. The results are that at one time, all programs may fit in memory, but sometimes later they will not. Since thrashing degrades system performance, it must be avoided; however, opinions vary as to what degree of system degradation represents excessive thrashing. Due to this variation of opinion, an option exists which allows setting of the thrashing detection point.

A resettable system parameter, AVAILMIN, allows the user to control the thrashing point of the system. This parameter, expressed in terms of percent of memory, can be set or altered at any time. After each overlay pass in the overlay control procedure, the amount of available main memory is compared with the amount of memory specified by AVAILMIN. When or if available space falls below the level set by AVAILMIN, the system will automatically suspend the lowest priority program and roll out this program's overlayable segments, indicating to the operator via a suspended by system message which program was suspended.

The operator can override the system when a program is suspended. The <mix-number> OK will cause the program to resume execution (but the system may immediately suspend it again). The suspended program which may resume processing and which may not be suspended again, may be terminated by entering <mix-number> DS. If it is mandatory that a suspended program resume processing and not be suspended again, the operator can increase the program's priority. The system will automatically resume any suspended program on a priority change.

The utilization of process plus I/O time as a time base was chosen in the calculation of WSSIZE since these times do not change while a program is suspended. This means a program's working set will not change during the time it is suspended. For this reason, the criteria used to resume a suspended program are based on its working set size (WSSIZE). Prior to the overlay pass in the overlay control procedure, the working set size of each program suspended is compared against the system available memory. The highest priority program whose working set size is less than the current amount of available

space will be resumed.

It can happen that suspending a program can result in some amount of main memory space being made available. Rather than let this space remain idle, the system may select a program from the schedule queue and place it into execution providing the program's core estimate indicates it will fit and there is no more than one program suspended. Since the automatic program suspension process is a part of the overlay control procedure, turning it off (setting the system overlay goal to zero) will also deactivate the automatic program suspension process. If overlay control is turned off while jobs are suspended, these programs are resumed before the procedure deallocates itself.

Control Programs

There are certain types of programs such as datacom MCS programs, data management control procedures, user-written control programs, etc., which must not be suspended or become scheduled at any time. For this reason, an operator message has been implemented which allows the operator to mark or unmark a program as a control program. A program marked as a control program will always go into immediate execution (i.e., never become scheduled). Additionally, if overlay control is turned on, it will never be selected for suspension.

Operator Messages for Overlay Control

The SF Message

The SF message is used to control and display the parameters associated with memory management. In response to an SF <ETX>, the system will display on the screen a message in the form of the following example:

1. OLAYGOAL = 25 % PER MINUTE

2. AVAILMIN = 7 % (XXXXXXX WORDS)

3. FACTOR = 100

4. MEM PRIORITY FACTOR = 5

The factor displayed by the SF <ETX> can be set by an SF <parameter number> <parameter value> <ETX> from the keyboard. For example:

SF 1 30 <ETX> would set OLAYGOAL = 30%

SF 2 5 <ETX> would set AVAILMIN = 5%

SF 3 80 <ETX> would set FACTOR = 80%

SF 4 5 <ETX> would set MEM PRIORITY FACTOR to 5.

The system allows the setting of any combination of one, more than one, or all parameters to be set with a single input; for example:

SF 1 25 2 5 3 100 <ETX> would cause:

PARAMETER 1 (OLAYGOAL) to be set to 25 % PER SECOND

PARAMETER 2 (AVAILMIN) to be set to 5%

PARAMETER 3 (FACTOR) to be set to 100%

Additionally, since operators are used to setting factor by entering SF <integer> <ETX>, an SF entry followed by a single value (i.e.,) SF 80 <ETX> will assume the parameter, FACTOR, is to be set.

The OG Message

The form of the OG message is:

<mix-number> OG <integer> <ETX>

This message allows uniquely setting the OLAYGOAL of each program in the mix. <mix-number> OF <ETX> will display the current value of OLAYGOAL.

The CU Message

The format of the response to the operator input message:

<mix-number> CU <ETX>

includes a system-maintained parameter in the form:

WSSIZE = <integer>

which will be displayed for both the job stack and the segment dictionary stack. The parameter, WSSIZE, is the working set of the program i.e., the size in words of the amount of memory resources this program has been using.

The CP Message

The operator message, CP, can be used to mark a program as a control program. A program marked as a control program will always be brought into the mix (never scheduled) and once in

the mix, it will not be selected for suspension. The format of the message is:

CP <reset-option> <program-name>

If the <reset-option> is left <empty>, the program will be marked as a control program.

Example:

(operator) CP SYSTEM/HARDCOPY

(system) SYSTEM/HARDCOPY CONTROL PROGRAM

If the <reset-option> is a minus sign (-), the program will be flagged as a non-control program.

Example:

(operator) CP SYSTEM/RJE

(system) SYSTEM/RJE CONTROL RESET

EXPLANATION OF PARAMETERS

OLAYGOAL Parameter

The parameter, OLAYGOAL, is used to set an overlay in advance rate. This OLAYGOAL, set by the SF message, is the overlay rate assigned to the MCP and system intrinsics. Additionally, it is used as a base to derive a default value based on priority for all programs entering the mix. The OG message can be used to override this default setting for a particular program.

The value of OLAYGOAL represents a percentage of a program's overlayable space which will be removed from core on a per minute basis. This tends to keep core free of unused segments, thus allowing more programs to be multi-processed.

AVAILMIN Parameter

The system monitors available core space in an attempt to determine when thrashing occurs. The user can set a system parameter, AVAILMIN, as a percentage of total core. The system monitors available space; and when it becomes less than AVAILMIN, it will automatically suspend the lowest priority job in the mix.

There are some exceptions, i.e., the following are never selected for suspension:

1. Any program marked as a control program;

2. Any program which has called sort;

3. That program which would reduce the number of running programs below the number of processors.

Even though a program is suspended, the system will continue to overlay its segments. Eventually, all of a program's overlay space will become available. Setting OLAYGOAL 0 will cause the system not to automatically suspend jobs. If the OLAYGOAL is set to zero while programs are suspended the system will automatically resume these programs.

SWAPPER

In DATA COMM situations, sometimes a program will have to wait on the output device because it is so slow (even 9600 BAUD is slower than a card reader), or the program may have to wait even longer for the terminal operator to enter a message (this may take minutes, or even longer the operator may be on a coffee break) and during this time the program may be totally inoperable. During this time, there is very little sense in tying up the memory with the program when something useful could be going on.

The first guess at a solution to this problem (in the context of the large systems) would be to overlay all of this program's overlayable memory space whenever the program goes into a state as described above. This has two drawbacks. The first is that a good deal of the memory of the program may be save memory and thus, would not be affected. The other drawback is that this would require many individual overlays with all of the inefficiency that that implies.

This last statement is not to imply that the memory management system on the large systems is inefficient. It is necessary to overlay any given memory area of any size, and this requires such a memory management system as exists. The point is that when you don't know what is going to happen next, you must be general. But, if you know that you are going to have to overlay an entire program, then the whole process must be re-thought.

Furthermore, many sequential disk accesses is an inefficient procedure, because the latency and or access time exceeds the data transfer time so that the effective data transfer rate is very slow. If you know ahead of time that many disk accesses must be made then they can be made much more efficiently at the same time. It is exactly this logic that leads one to block tape and disk files.

It is clear, then, that the problem is to read and write the

whole program and all of its memory all at once (if possible, in one disk access). This is exactly the process that is used on the large systems and it is called swapping, meaning that two or more jobs (tasks) are swapped between disk and memory.

In fact, there are more than just data comm in which a swapping environment is desirable. Anytime a task is suspended and cannot operate for long enough period of time to make swapping reasonable, it should not tie up memory. The reason for the ambiguous qualification here is that during a tape or any other I/O operation with only one buffer (or all empty buffers on a read, or all full buffers on a write) the task is inoperative, but there is no sense in swapping it out. Clearly there must be some reasonable wait time. I/O is slow compared to process time, but not that slow.

In general, swapping will be valuable whenever:

1. A job may be suspended indefinitely. This includes at least these cases:

   a. St (opped)

   b. Waiting for a necessary file to become present.

   c. Waiting for operator action.

2. Whenever a data comm file is present.

SWAPPING, Implementation

In order to let parts of the system operate normally, not all tasks are subject to swapping. These tasks include:

1. Independent runners.

2. Utilities such as AUTOBACKUP.

3. MCS's.

To do this, a space of memory is set up as the ""SWAPSPACE."" within this space, tasks are swapped. Outside of this space, all goes as usual.

Each task that is assigned to swap space is allocated a contiguous piece of memory within the swap area. This way the entire task may be swapped out at once. While it is in its memory slot, it uses normal memory management, but only within the slot. This has the added benefit that one task's excesses do not affect any other tasks.

Within each task's swap space the first two words are used exactly like MEMORY[MEMBASE] and MEMORY[MEMBASE+1] but only

for the task's swap space.

Within each stack, outside of swap space, that is associated with a stack in swap space, is a word called MEMLOC. If the stack is in swap space (and not on disk) then this word is like a descriptor specifying the stack's swap space (address and length). If the stack is not in swap space then this word is like the M descriptor, specifying all of memory. Thus GETSPACE and FORGETSPACE use this word to determine what two words to use for MEMORY[MEMBASE] and MEMORY[MEMBASE+1]. Also, within each PIB is a word called SWAPDATA. For each stack in swap space, this a descriptor pointing to the information for this stack. Something must be left in memory to specify where on disk this swapped stack is located.

The whole swap space is divided up into multiple 990 word slots. Every task running in swap space is allocated some integer number of slots.

Whenever a task spends more than 10 % of its elapsed time in the swap space overlaying, it is considered to be thrashing, and will be swapped in the next time with a larger number of swap slots. When the task's memory is (for any reason) to be swapped out to disk, it is written with one disk I/O into a disk file called ""SYSTEM/SWAPDISK."" When it is swapped back into main memory, it will be put into any available portion of the swap space that is large enough to contain the number of swap slots necessary. This memory space may not be the same as it was before, so the MCP must fix up all of the descriptors and memory links.

The MCP procedure that does all of the I/O to and from disk and manages the swap space and fixes up the address, etc, is called SWAPPER and is forked when the ""SW"" message is entered (assuming it is not already running). The swapper checks on disk for a file called in segment 0 of the file. This first segment contains swap parameters such as the number of 990 word memory slots to be used. SWAPPER uses GETSPACE to try to get this much save memory. If it cannot get it, it displays a message and leaves the mix.

SWAPPER QUEUES

SWAPQ Queue

WORD 1: Linkage.

WORD 2: Stack number.

SWAPOUTREQUEST and SWAPINREQUEST are the two procedures which place entries into this queue and cause SWAPREQUEST which is the event on which SWAPPER waits. SWAPOUT for:

1. Read DATA COMM file and no input available.

2. Write DATA COMM file and buffers full.

3. ST'ed

4. ACCEPT

5. TIME SLICE expired.

6. GETSPACE cannot find space.

SWAPPING AND TIME SLICING

The purpose of job swapping or time slicing is to provide a means of more efficiently using memory in a datacom environment. One half of the objective is accomplished by organizing the memory space for each swap-job in such a way that all of the parts of the job that can be overlayed (swapped) are in one contiguous area of memory. This allows a single disk operation to write the entire area to disk (thereby freeing the memory area) or a single disk read to return the entire area to memory for resumption of processing. The remainder of the objective is reached by allowing a large number of burst-oriented task to run without freezing memory resources during their dormant periods with their use of the subspace task execution option (time slicing). Memory is freed by swapping the dormant task to disk so that its memory resources are available for use by another such task. Because a large number of tasks are bidding for a somewhat smaller memory resource, tasks, which for some reason, discontinue their burst-orientation for some duration of time must have an artificial burst rate imposed upon them. This is timing slicing and is necessary to ensure that all tasks will have an opportunity to use the subspace memory resource.

The area of memory which is to contain swap-jobs is referred to as the swap-area. The size of the swap-area is installation specified and can be located anywhere in main memory. Some of the characteristics of jobs in the swap area are as follows:

A. The amount of memory allocated for a swap-job is in multiples of 990 words, and this is calculated from the memory estimate of the job when it is initiated. It should be noted that the amount of memory is rounded up to the next multiple of 990 words.

B. The visible name of the swap-job has a "#" between the priority and its name as displayed on the screen.

C. A task operating within a subspace will be swapped out to disk:

1. Whenever the output buffers for files being directed to the datacom subsystem are filled and the task attempts more such output;

3. Whenever the task has been suspended;

4. Whenever the time slice allocated the task has expired;

5. Whenever the task exceeds the subspace size allocated to it on its previous swap-in.

D. Tasks are returned to memory (with relocation so that it is not required that identical memory space from which a task was swapped be available in order to swap the task back to core) whenever they are capable of utilizing the processor and there is adequate memory available for the swap-in. Memory assigned to a subspace task will be increased up to the size of the subspace whenever a task exceeds the currently allocated space. The swap-in algorithm attempts to pack the task towards one end of the swap area to avoid checkerboarding.

There are two priority levels for selecting ready-to-run tasks for swap-in:

1. Demand swapping

2. Time-sliced tasks

Demand Swapped Tasks are tasks which are new to the system, which have received the datacomm input for which they are waiting, tasks for which the datacom subsystem has output at least one half of the data which excess originally caused the swap-out, and tasks which have been swapped because of time slice expiration. The exception to this is that whenever the number of tasks inserted in front of a slice job exceeds its slice number by two, that task will revert to demand status. Within demand status, tasks are ordered in a first-in first-swapped order.

Time Sliced Tasks are tasks swapped out because they exceeded their time slice and will be swapped into available memory only if there are no demand status swap requests. It should be noted that priority is not a direct consideration in the swapping algorithm; Instead, it appears in the formula used to compute time slices. The time slice allocated a task is expressed in terms of both elapsed time and processor time. Before allocating a processor to a swappable job, its process processor and elapsed time slices are checked. If either is exceeded, a new slice is computed as per the time slice formula, and the job is swapped out. When a job is swapped out due to a demand condition (i.e., datacom input required),

its slice number is reset to zero.

Each time a task is swapped because of time slice exceeded, the slice number is incremented by one. This number is subject to a maximum value as specified in the swapdisk file, with a system imposed maximum of 256. The formula for computing a time slice is:

```
T = (N * K1 + C + P DIV 8) * K2 + M * 416667

E = T * R
```

Where:

T is the processor time slice, units are 2.4 microseconds.

E is the elapsed time slice.

N is the slice number (the maximum value for N is obtained from word 5 of the first record of swapdisk. If the word is 0, a default value of 7 will be used).

C is the core space used by the task in chunks.

M is the minimum time slice in seconds. This number is obtained from word 4 of the swapdisk. A default value of 1 second will be used if this word is zero.

K1 is 8.

K2 is 5000.

P is priority.

R is the ratio of elapsed time to processor time. This number is obtained from word 6 of swapdisk. If the value of word 6 is 0, then 2 is used; otherwise, word 6 must be greater than or equal to 1.

E. The PIB variable of each swap-job is kept as a non-overlayable item outside of the swap-area such that the various task attributes may be investigated whether the swap-job is in or out of memory.

F. Henceforth, by default, all tasks run by CANDE are run in subspace if SWAPPER is running. Compilers and system utility routines are run with SUBSPACES = 1 (SWAPREENTRANT), so the D1 stack is in normal memory and the D2 stack is in a subspace. User programs are run with SUBSPACES = 2 (SWAPSTANDARD): D1 and D2 stacks (code and DATA) are both in the subspace if the code file is in a usercode library; if the code file is in the SYSTEMDIRECTORY then the code is reentrant and data is

swapped. The execution part of a compile-and-go RUN command has SUBSPACES = 3 (SWAPALL): code and data are both in subspace.

A new secondary control statement (modifier) may be used to override the default settings. The syntax and semantics are:

SUBSPACES = 0 Do not use subspace

SUBSPACES = 1 Data in subspace

SUBSPACES = 2 Data in subspace, code also if user program

SUBSPACES = 3 Data and code in subspace

(The equal sign is optional.)

Examples:

```
COMPILE; C SUBSPACES = 0 [Compile without swapping]

E SYSTEM/UTILITY; SUBSPACES = 1; [Swap data, not code]
```

Note that subspace setting for execution may not be specified at compile time (with either compile or run).

G. Once initiated, a task is either in the swap-area or it is not; furthermore, it will not be changed from swappable to non-swappable, or vice-versa.

H. SWAPPER does not remember the prior position of a swap-job on disk. On a swap-out, swapper allocates a place within SYSTEM/SWAPDISK to which the swap-job will be written. Note that this allocation is done on a rotational basis among the rows of SYSTEM/SWAPDISK. The purpose is to distribute the disk traffic among the EUs if the rows had been distributed by the installation, either by classing or by using IADMAPPER.

SWAPPER Process

The actual swapping process of the MCP is accomplished by a visible independent runner named SWAPPER. SWAPPER waits on an event called SWAPREQ which is caused when a new request is given to SWAPPER or when a condition has changed which is of interest to SWAPPER.

SWAPPER is initiated by the keyboard request, SW. (Note that this is the sole input required to produce the swapping environment.) Upon initiation, SWAPPER checks for the

availability of a file on head-per-track disk or System-Resource- Pack, called SYSTEM/SWAPDISK. This file is the disk area to and from which tasks are swapped. Also contained in this file are nine parameters needed by SWAPPER. One parameter is the size of the swap-area and two of the nine parameters contain information concerning the physical attributes of the disk file itself. Four parameters contain information pertaining to time-slicing, and may be changed dynamically while SWAPPER is executing by means of the ACCEPT system input message. The last two parameters are associated with EXPRESS jobs.

In order to perform swapping SYSTEM/SWAPDISK must exist. This file is created by a program and can be changed by ODT commands. A program to create SYSTEM/SWAPDSK can be found in the SOG, Volume 2, Section 16 (form number 5001688).

Parameter Definitions

THE SWAP-AREA SIZE PARAMETER IS:

CORESIZE: This parameter is an integer which defines the SWAPAREA allocation in units of 990 word memory slots.

THE DISK FILE PARAMETERS ARE:

DISKSIZE: This parameter is an integer which defines the size of a row of swap disk in units of 44 segment (30 words each) disk slots.

DISKROWS: This parameter is an integer which defines the number of swap disk rows.

THE TIMESLICING PARAMETERS ARE:

MINTIME: This parameter defines the minimum processor time allotted to a swap job. The value is specified in units of seconds, and can be a decimal number. If a value of zero is supplied, a default of 1 second is used.

MAXSLICENUMBER: This parameter is best described in the following context. Each time a job is swapped because its processor time or elapsed time has been exceeded, its slice number is incremented by 1, and a new time slice is computed. The MAXSLICENUMBER parameter is an integer which defines the maximum value of the slice number. If a value of 0 is supplied, a default of 7 will be used.

RATIO: This parameter defines the ratio of the elapsed time that a swap job may reside in core to its current processor time slice. The value supplied can be a decimal number. If a value of 0 is supplied, a default of 2 will be used.

MAXCORE: This parameter defines the maximum subspace size (in 990 word chunks) which will be allowed for a job. SWAPDISKMAKER multiplies the value of this parameter by 990 to the number of words and stores it in word seven record zero in SYSTEM/SWAPDISK. If SWAPPER finds the value of word seven to be zero, then the MAXCORE is set to the minimum of the CORESIZE of 130,680 words.

THE EXPRESS PARAMETERS ARE:

EXPRESERVE: This parameter is used to specify how many 990 word core slots should be used for the EXPRESS area within SWAPSPACE.

EXPMAXCORE: This parameter is used to specify the maximum size, in core slots, a task may have and still be able to

acquire EXPRESS status.

EXPTIME: This parameter is used to specify the maximum time slice for for an express task.

THE PROCESSOR QUEUEING PARAMETERS ARE:

PRIORITYBIAS: This parameter specifies the effect (bias) of a tasks priority on queueing.

UTILIZATIONBIAS: This parameter specifies the effect of interactivity on queueing. A task is interactive as long as its slice number is less than maxslicenumber.

MEMORYBIAS: This parameter specifies the effect of memory size on queueing.

IOBIAS: This parameter specifies the effect of the number of I/O's required to swap in a multiprocess family.

MEMORY MANAGEMENT, ANOTHER LOOK

The memory management system has the following features:

COMPLETELY DYNAMIC

VIRTUAL MEMORY

VARIABLE LENGTH AREAS

WORKING SET CONCEPT

EFFICIENCY (MINIMIZE CHECKERBOARDING)

SPEED

MEMORY PROTECTION

In order to obtain speed, the memory management system links available areas together with memory links which are words constructed so that the linked list lookup (LLLU) operator may be used to find available areas. These words proceed and follow all memory areas. This gives the memory protection as a side effect because the memory links are protected words (bit 48 is on).

For available areas, the memory links are named:

AVAILA

AVAILB

(AVAILABLE AREA)

AVAILY

AVAILZ

There are four links around in use areas. These links are not actually linked to each other. The in-use links are given the following names.

LINKA

LINKB

LINKC

(IN-USE AREA)

LINKZ

In order to avoid checkerboarding the save areas (memory areas that are not overlayable and can not be moved) are kept adjoining and as close together as possible. This is important because save areas tend to move around less (be less volatile) than overlayable areas. This implementation will avoid the following type of problem:

> two save areas close together with a small area between them

This is undesirable because the save areas may not get moved for a long time and the small area may be too small to be usable in most cases, thus it is effectively wasted memory.

The way this is implemented is to link the available areas into two different lists one list of areas that are most efficiently used for save memory, and the other list of areas that will be most efficiently used as overlayable areas.

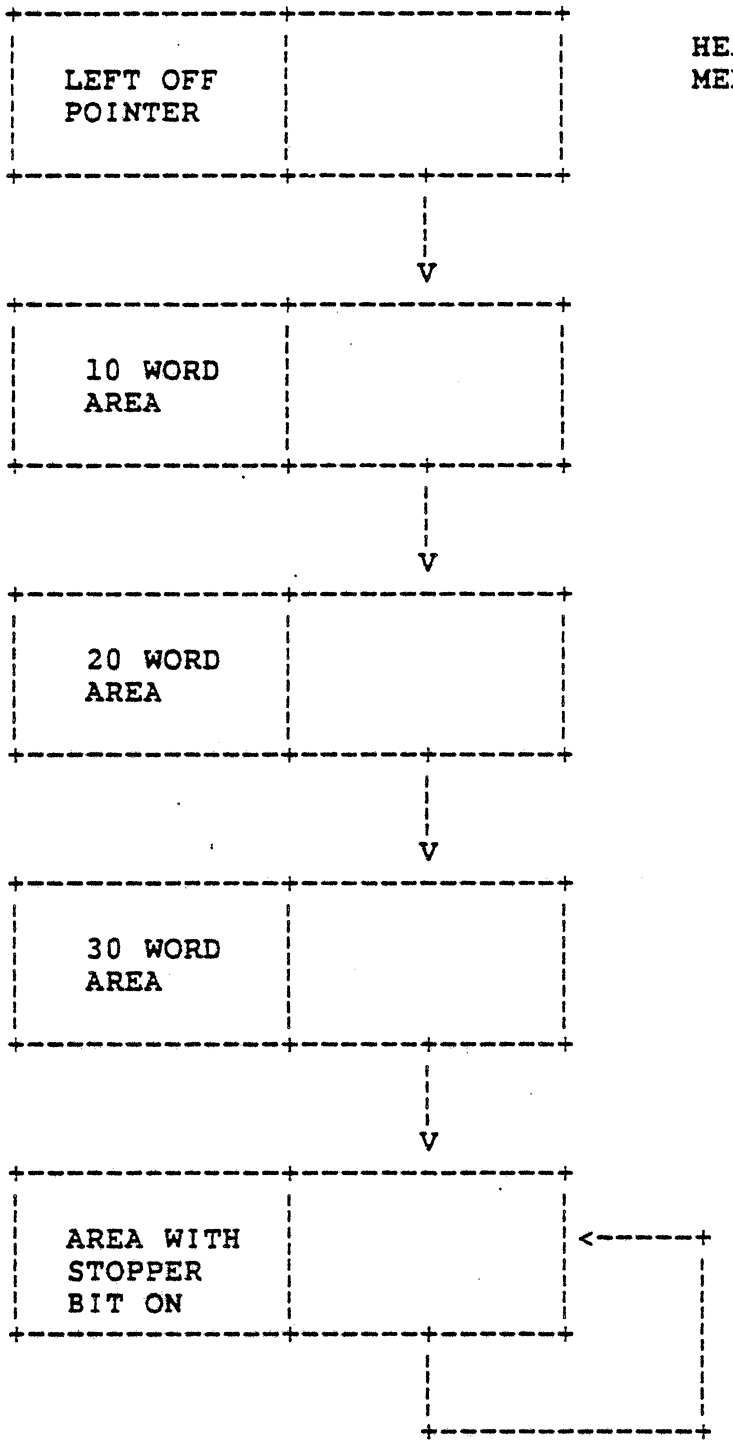
In particular, an available area is linked into the (to be used as) save list if it follows a save area (figure 7-2). An available area is linked into the (to be used as) olay list, if it adjoins an overlayable area (figure 7-3).

Notice that with these two rules all available areas will be linked into at least one list (necessary), and that an available area might be linked into both lists (desirable). See figure 7-4. The reverse links B and Y are used as back pointers. See figures 7-5 and 7-6. Notice that although the A-B and Y-Z links are associated with the same area, they never actually point to each other. One other difference between the two linked lists is that the save list is linked in order of size, smallest first, while the olay list is linked in order of age, most recent first. Thus, a best fit is used when obtaining a save area (to avoid possible checker-boarding) but a first fit is used when obtaining an overlayable area (for speed).

Two words of memory pointed to by the MEMBASEPLACE word in the BOXINFO array are used for starting points of the two lists. The list at MEMORY[MEMBASE] is used for overlayable areas. The list at MEMORY[MEMBASE+1] is used for save areas. The word at MEMORY[MEMBASE-1] is used as a lock for the links. See figure 7-7 and 7-8 for a layout of these links. There is one set of links for each box configured. Thus, on a monolithic system there are two chains (one set). A tightly coupled 2 processor system will have 3 sets of links (local box 1, local box 2, and Global). In addition, if swapper is running there is a set of chains in each users swap area (the set of links is pointed to by the MEMLOC word in the stack). In all cases the links look and function the same. See figure 7-7 and 7-8.

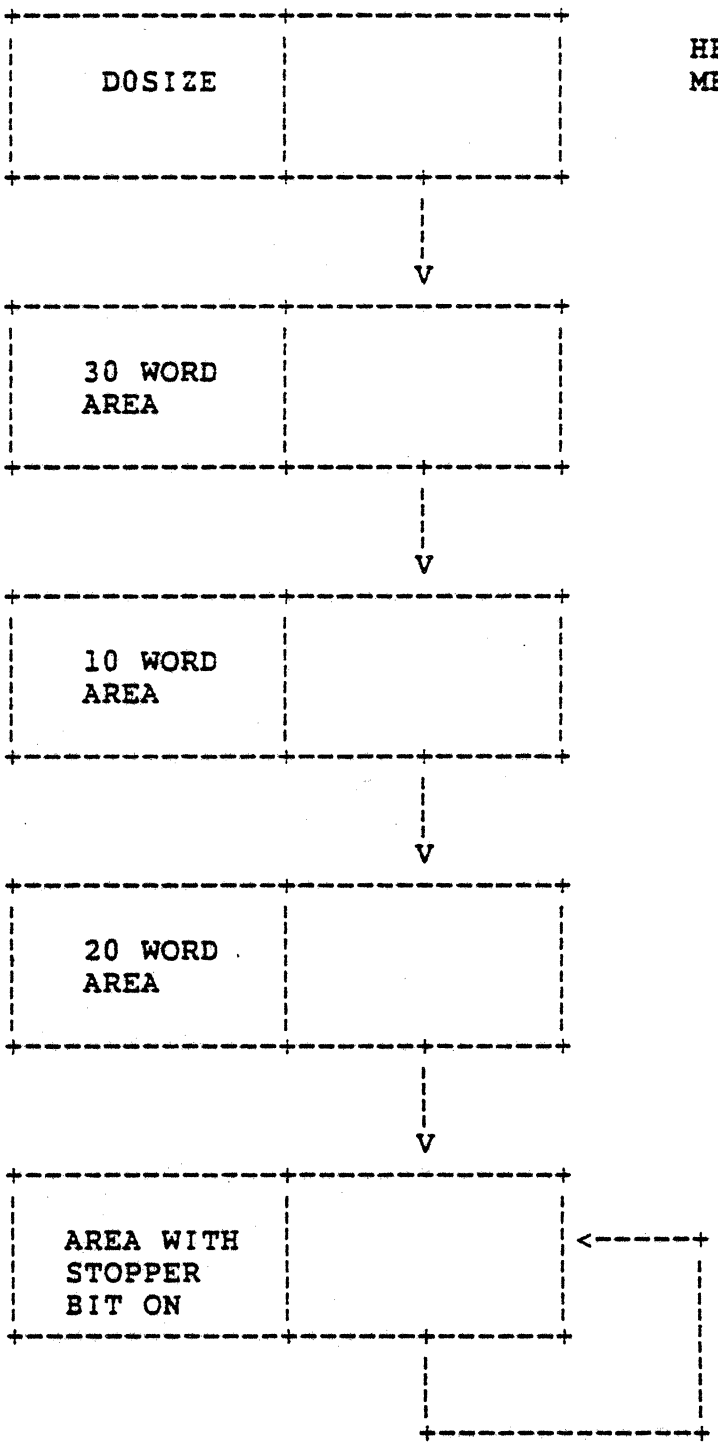
Most major memory management is done by the procedures GETSPACE and FORGETSPACE. However, some special purpose memory is handled by GETAREA and FORGETAREA. GETAREA is used

for highly volatile fast turnaround save memory areas.



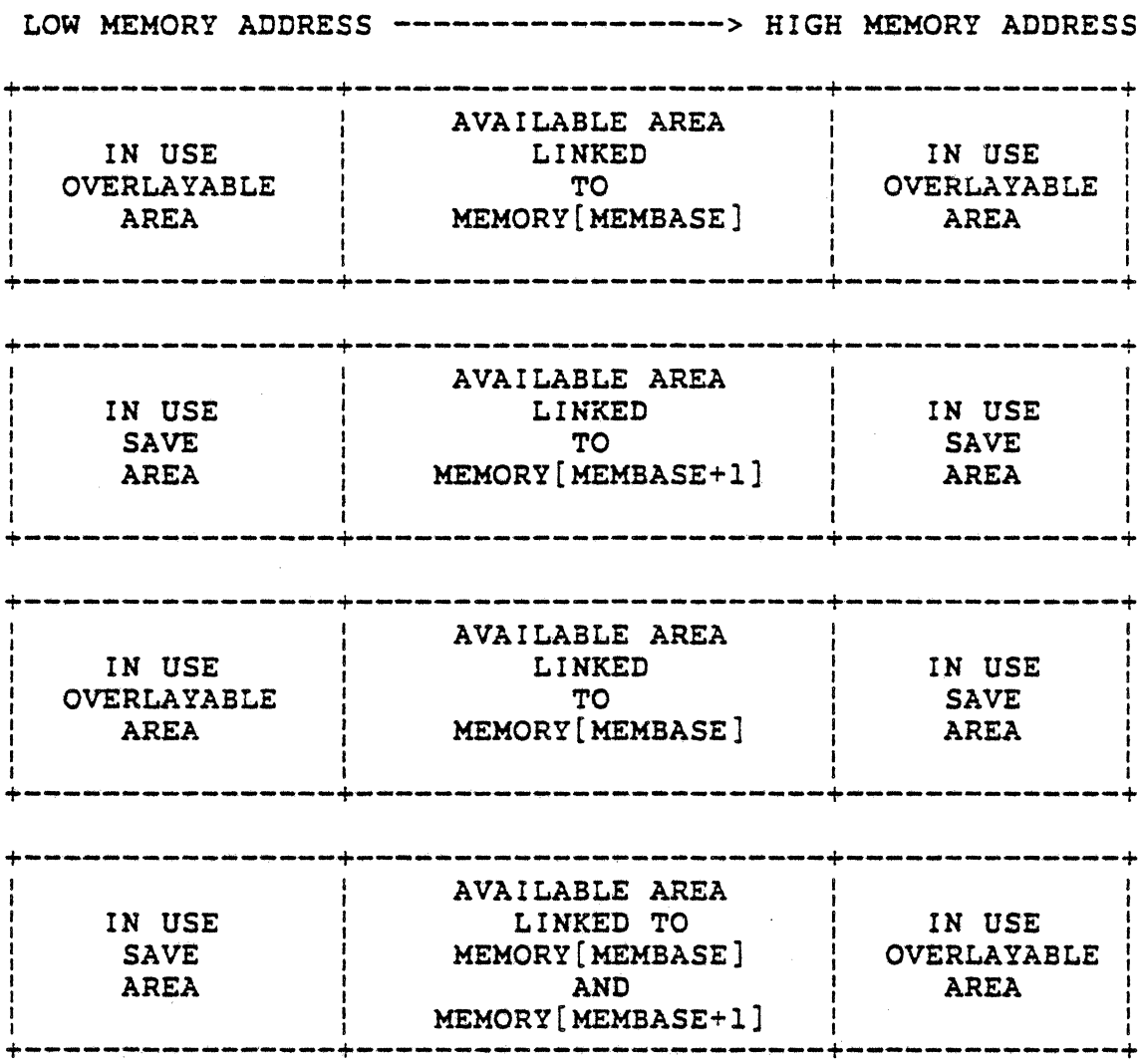
SAVE MEMORY LIST. ORDERED BY SIZE. SMALLEST TO LARGEST.

Figure 7-2. AVAILZ Links



OVERLAYABLE MEMORY LIST. ORDERED BY AGE. LAST AREA FORGOTTEN IS FIRST IN LIST.

Figure 7-3. AVAILA Links



MEMORY MANAGEMENT WILL TRY TO KEEP ALL SAVE MEMORY AT THE LOW END OF MEMORY. ALL OVERLAYABLE MEMORY WILL BE KEPT AT THE HIGH END OF MEMORY. THIS IS DONE TO EASE MEMORY CHECKERBOARDING.

Figure 7-4. Memory Links

GETSPACE

This procedure is called to find memory space. It will return the address of the area if one is found. If one is not found, one of several things will happen, including returning 0.

GETSPACE has five parameters:

1. SIZE size of area needed (not including links).

2. MOMSNR Stack number of MOM descriptor.

3. TYPE Specifies save or non-save memory and other fields.

4. MOMADDR Absolute address of MOM descriptor.

5. MEMLOC Word which specifies which set of links to use.

GETSPACE TYPE parameter:

DONTLOSEMEF GETSPACE sets this bit when it calls itself for core to core overlay. If this bit is set and GETSPACE cannot find the area in one pass through the correct list, then it will return 0, and make no attempt at overlaying. This is to prevent recursion.

ICANDOWITHOUTITF This is set when non-critical areas are being gotten and the caller can do without the area. GETSPACE will overlay but will return 0 if it cannot find the area.

IWONTTELLIFYOUWONTTELLF This is set when the caller can wait for the area, but must have it. In this case, GETSPACE will try to overlay and if that fails, will wait on MEMFREE and then go try again.

If none of the three bits (DONTLOSEME, ICANDOWITHOUTITF and IWONTTELLIFYOUWONTTELLF are on and GETSPACE cannot find the memory space after overlaying then a no memory condition will arise.

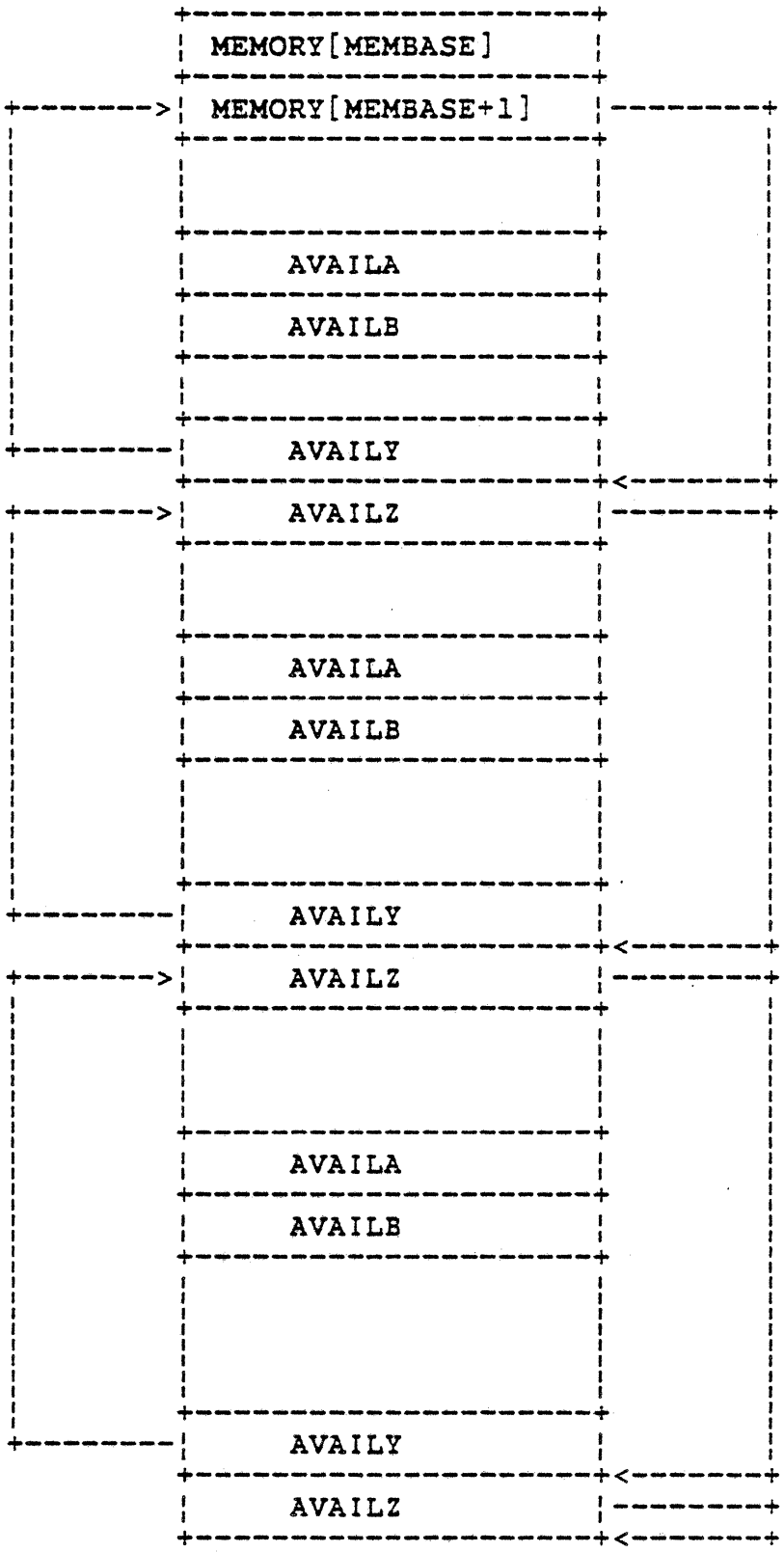


Figure 7-5. AVAILY and AVAILZ Links

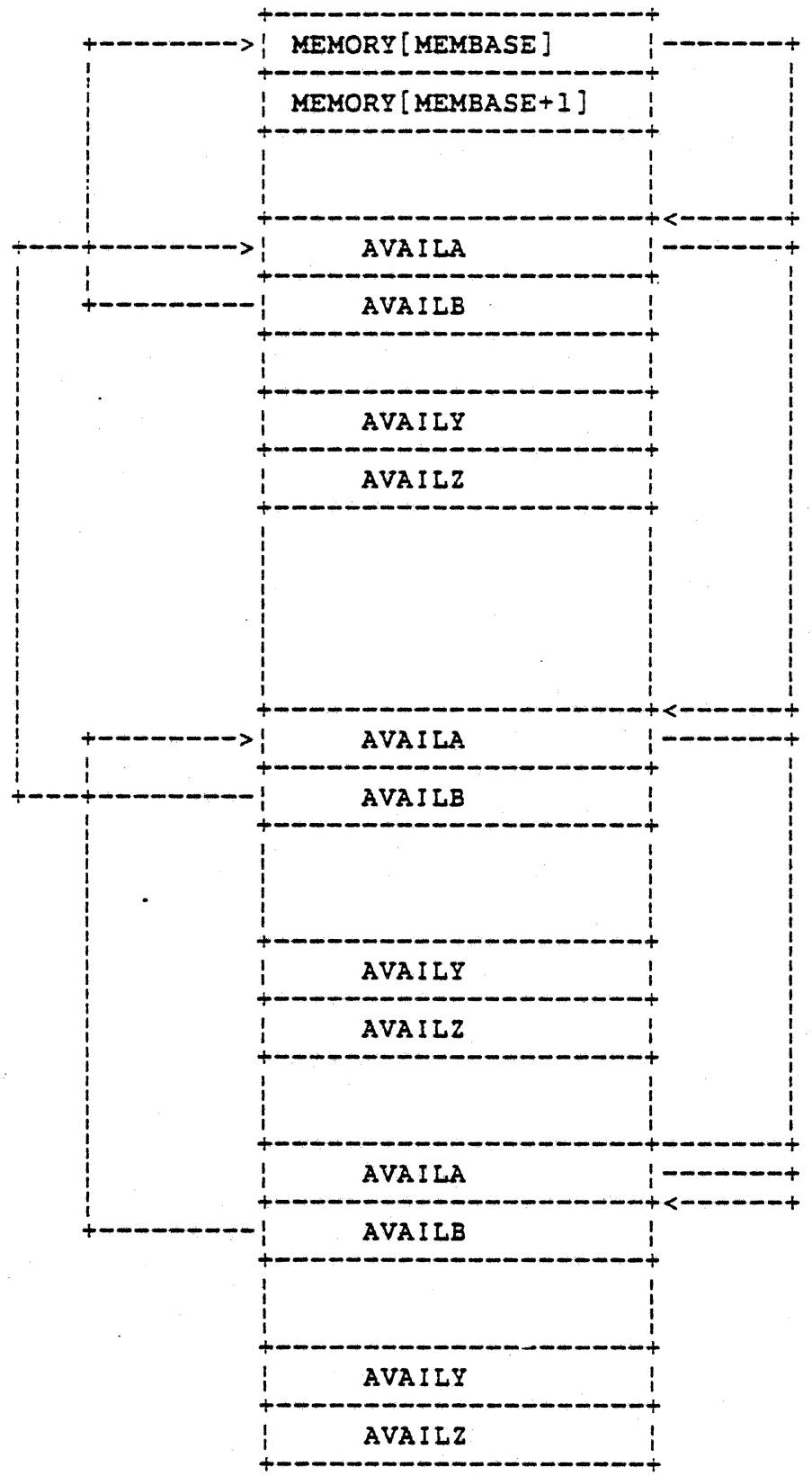
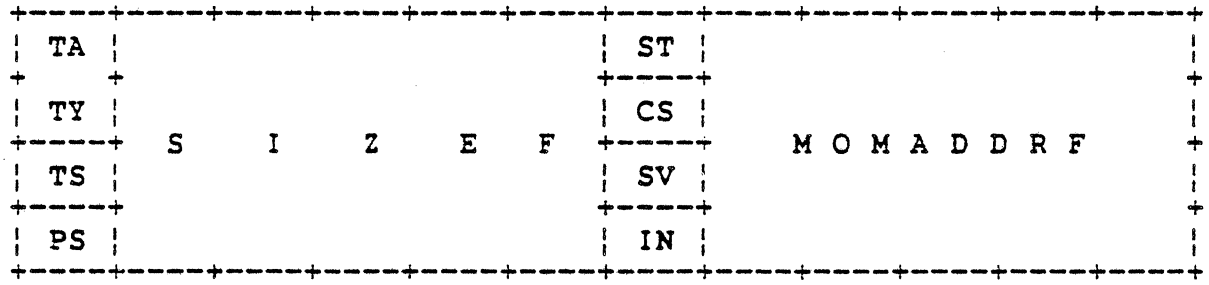


Figure 7-6. AVAILA and AVAILB Links

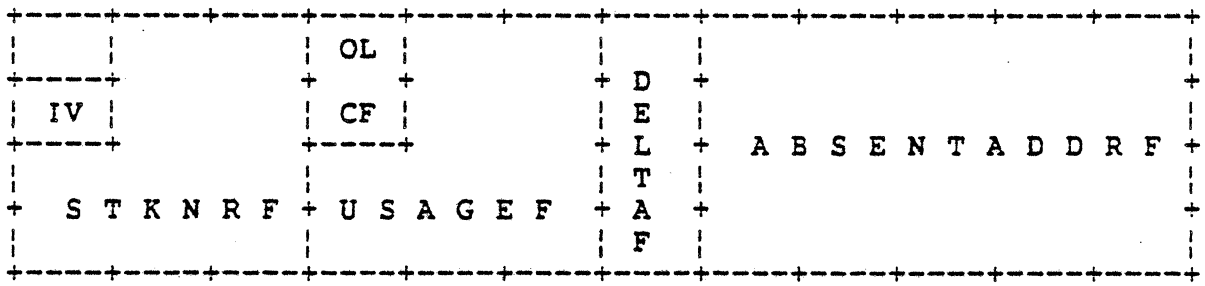
L I N K A T A G 6



[47:2] TAG6TYPEF
[45:1] TEMPSAVF
[44:1] PRECURSAVF
[43:20] SIZEF
[23:1] STICKYF
[22:1] CURSAVF
[21:1] SAVEF
[20:1] INUSEF
[19:20] MOMADDRF

TAG6TYPEF THIS FIELD HAS THE VALUE 3.
TEMPSAVF IF AREA IS FROZEN
PRECURSAVF AREA WAS CURSAV BEFORE FREEZING
SIZEF SIZE OF THE AREA. INCLUDES LINKS.
STICKYF AREA CAN BE MOVED BUT NOT OVERLAYED.
CURSAVF AREA IS CURRENTLY SAVE.
SAVEF AREA IS PERMANENTLY SAVE.
INUSEF AREA IS IN USE.
MOMADDRF ADDRESS OF MOM.

L I N K B T A G 7

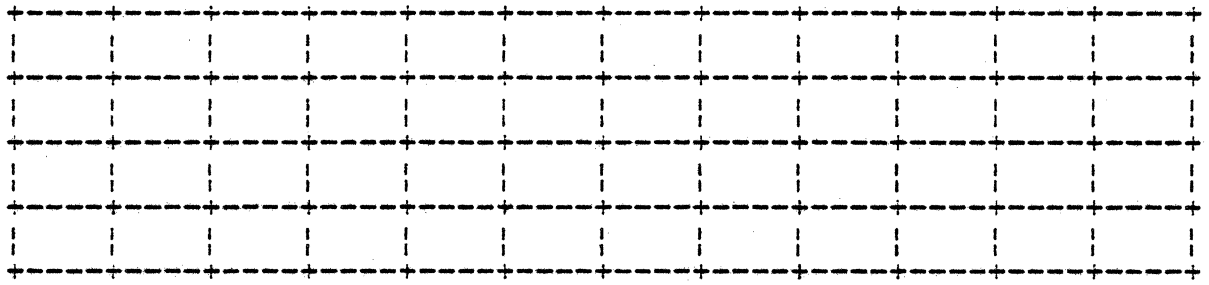


[46:1] INVALIDSTKN
[45:10] STKNRF
[35:2] OVERLAYCF
[33:10] USAGEF
[23:4] DELTAF
[19:20] ABSENTADDRF

INVALIDSTKN MUST BE SET TO 0.
STKNRF OWNER STACK NUMBER
OVERLAYCF 0 NORMAL. 1 TO BE OVERLAYED.
2 TO BE FORGOTTEN NOT OVERLAYED.
USAGEF USE OF AREA (FIB STACK SIB).
DELTAF WORDS IN AREA NOT ALLOCATED TO USER.
ABSENTADDRF ABSENT DESCRIPTOR ADDRESS. THE OVERLAY ADDRESS.

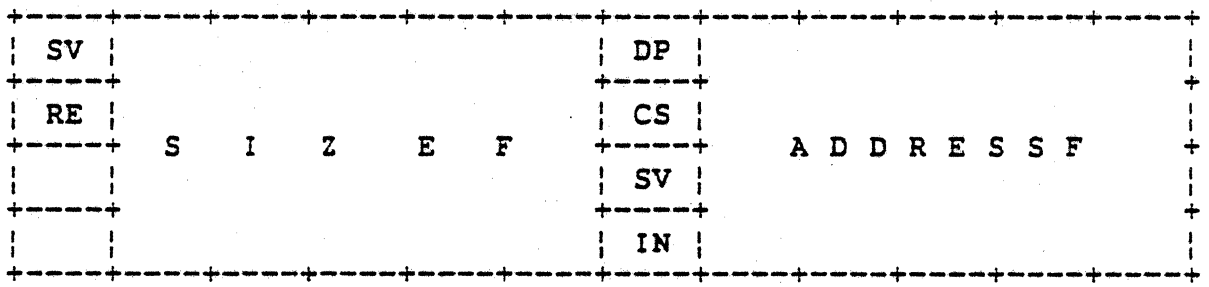
Figure 7-7. In use Links (1 of 2)

L I N K C T A G 3



GENERAL USE AREA FOR THE FOLLOWING THINGS:
RCW FOR CODE THAT MADE ARRAY PRESENT.
SPECIAL INFORMATION FOR DIRECT ARRAYS.
IOCW FOR OVERLAY.

L I N K Z T A G 3

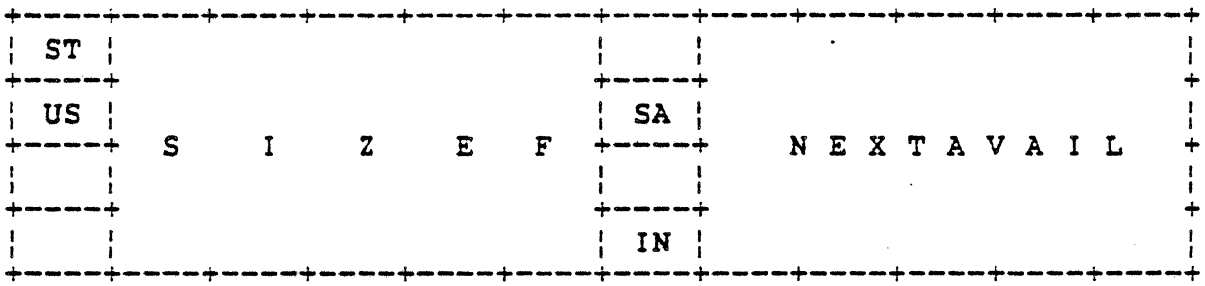


[47:1]	SAVINGF	[22:1]	CURSAVF
[46:1]	DONTRESIZEF	[21:1]	SAVEF
[43:20]	SIZEF	[20:1]	INUSEF
[23:1]	DELTAPF	[19:20]	ADDRESSF

SAVINGF	AREA IS BEING MADE UNAVAILABLE. BEING SAVED.
DONTRESIZEF	AREA MUST NOT BE RESIZED.
SIZEF	SIZE OF THE AREA. INCLUDES LINKS.
DELTAPF	THERE ARE DELTA WORDS IN THE AREA.
CURSAVF	AREA IS CURRENTLY SAVE.
SAVEF	AREA IS PERMANENTLY SAVE.
INUSEF	ON IF AREA IS IN USE.
ADDRESSF	MEMORY PRIORITY TIMER.

Figure 7-7. In use Links (2 of 2)

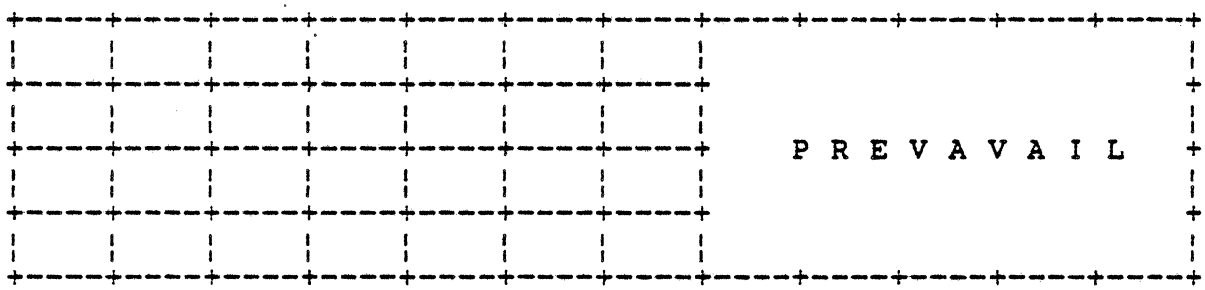
A V A I L A T A G 1



[47:1] STOPPERF
[46:1] USABLEF
[43:20] SIZEF
[22:1] SAVEDF
[20:1] INUSEF
[19:20] NEXTAVAIL

STOPPERF — ON IF THIS IS THE LAST AREA. THIS WILL FORM A LARGE SIZE AND STOP THE LIST LOOKUP OPERATOR.
USABLEF — ON IF AREA CAN BE USED. OFF IN LAST AREA.
SIZEF — SIZE OF THE AREA. INCLUDES LINKS.
SAVEDF — ON IF AREA IS BEING MADE UNUSABLE (SAVED)
INUSEF — ON IF AREA IS IN USE.
NEXTAVAIL — LINK TO NEXT AVAILABLE AREA IN OVERLAYABLE CHAIN.

A V A I L B T A G 0

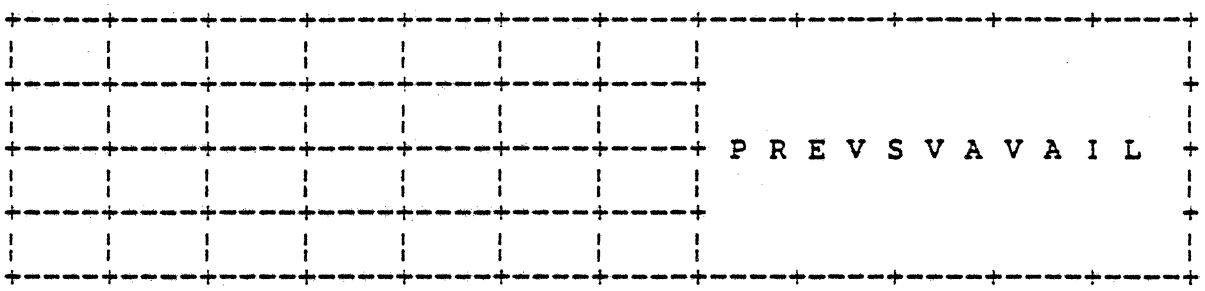


[19:20] PREVAVAIL

PREVAVAIL — LINK TO PREVIOUS AVAILABLE AREA IN OVERLAYABLE CHAIN. THAT IS, A BACK LINK.

Figure 7-8. Available Memory links (1 of 3)

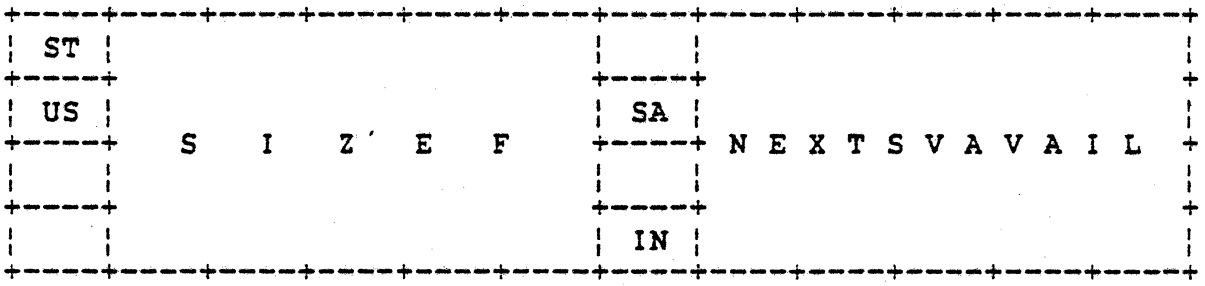
A V A I L Y T A G 0



[19:20] PREVSVAVAIL

PREVSVAVAIL LINK TO PREVIOUS AVAILABLE AREA IN SAVE CHAIN.
THAT IS, A BACK LINK.

A V A I L Z T A G 1

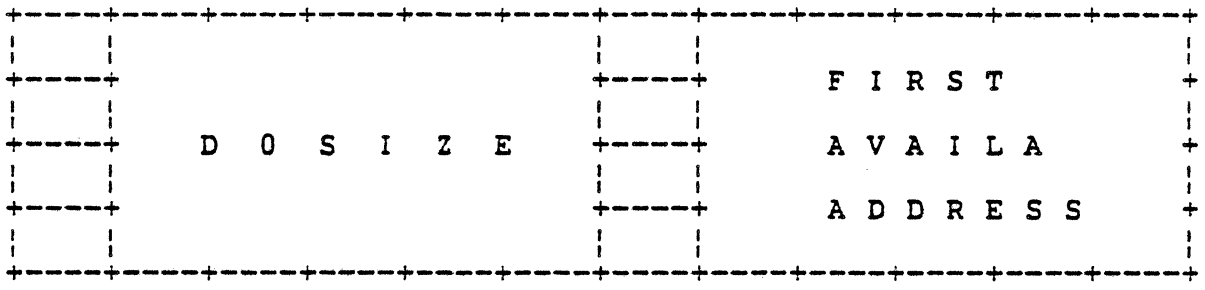


[47:1] STOPPERF
[46:1] USABLEF
[43:20] SIZEF
[22:1] SAVEDF
[20:1] INUSEF
[19:20] NEXTSVAVAIL

STOPPERF ON IF THIS IS THE LAST AREA. THIS WILL FORM A LARGE SIZE AND STOP THE LIST LOOKUP OPERATOR.
USABLEF ON IF AREA CAN BE USED. OFF IN LAST AREA.
SIZEF SIZE OF THE AREA. INCLUDES LINKS.
SAVEDF ON IF AREA IS BEING MADE UNUSABLE (SAVED)
INUSEF ON IF AREA IS IN USE.
THAT IS, A BACK LINK.
NEXTSVAVAIL LINK TO NEXT AVAILABLE AREA IN SAVE CHAIN.

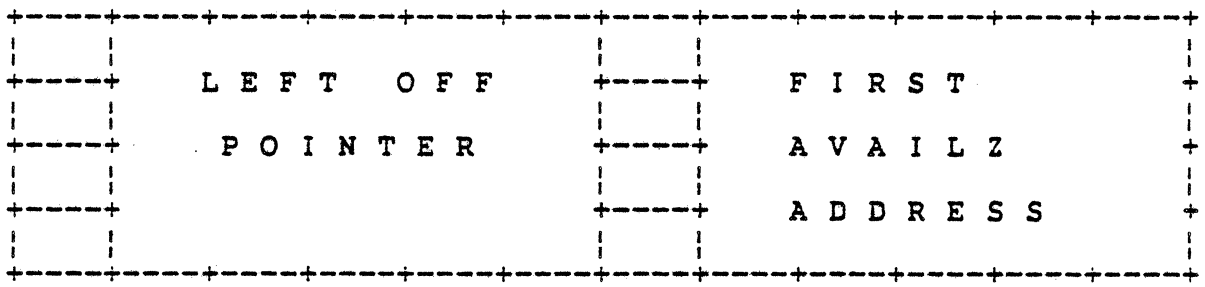
Figure 7-8. Available Memory links (2 of 3)

OVERLAYABLE CHAIN HEAD WORD



HEAD WORD IS LOCATED AT MEMORY[MEMBASE]

SAVE CHAIN HEAD WORD



HEAD WORD IS LOCATED AT MEMORY[MEMBASE+1]

Figure 7-8. Available Memory links (3 of 3)

FORGETSPACE

This procedure is called to return in use spaces that were obtained by GETSPACE. Its single parameter is the address of the space to be forgotten or a data descriptor to that space. Unlike GETSPACE, FORGETSPACE always successfully completes its task, so it returns no value.

The main feature of FORGETSPACE is that it always consolidates available areas, to give back the largest possible area. This is done by checking the preceding and following areas. If either one or both is available, then the two or three areas are consolidated into one available area. Also, any DELTA WORDS in the preceeding area are consolidated into the available area.

FORGETSPACE links this consolidation maximum size available area into the correct available lists. See figure 7-4.

GETAREA and FORGETAREA

GETAREA and FORGETAREA are called to get and forget small or special purpose areas. The types of areas handled by this special process include at least the following:

DATA COMM MESSAGE AREAS

MESSAGES FROM THE MCP TO CONTROLLER

ETERNALIR QUEUE ENTRIES

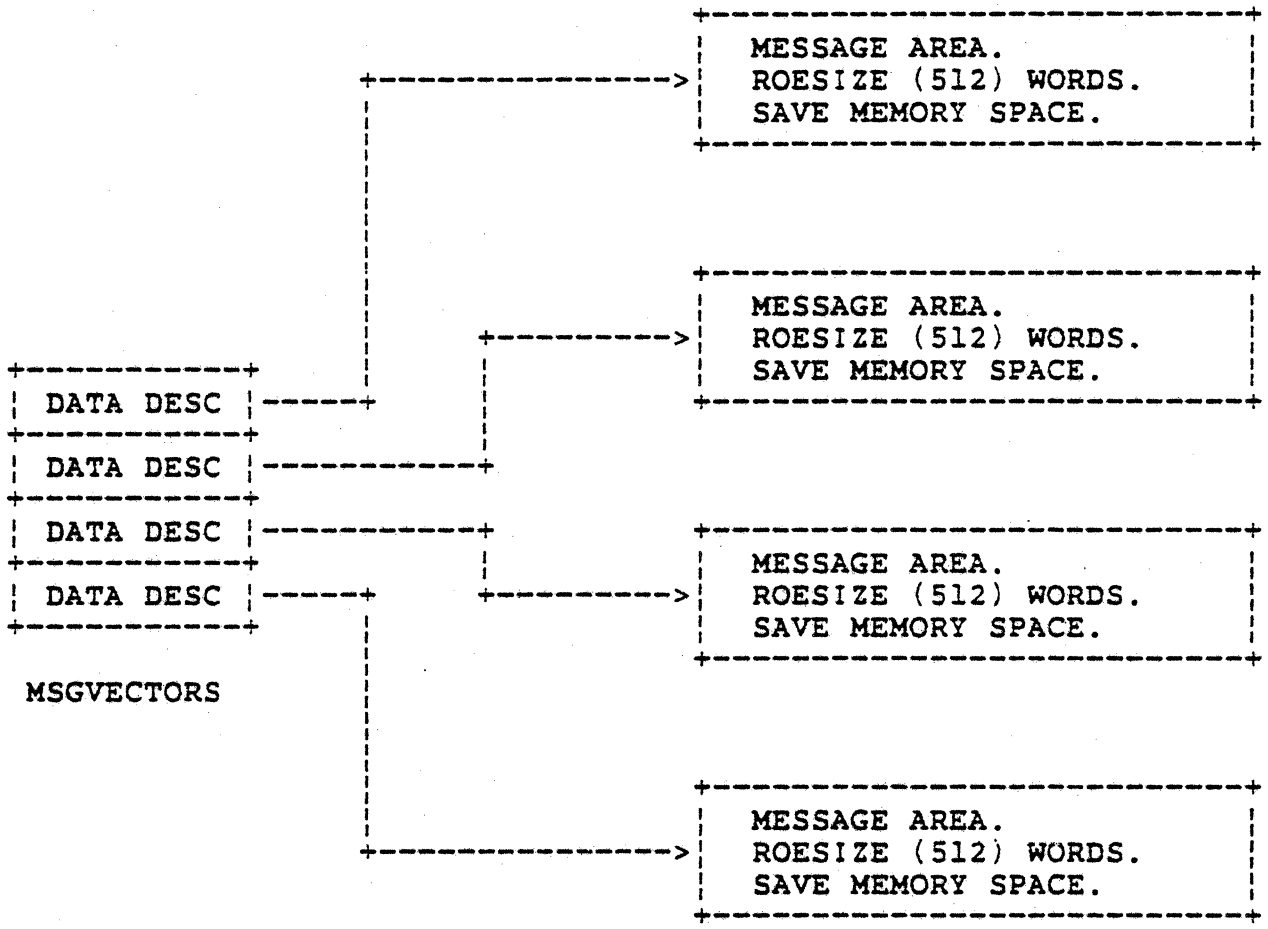
FORK QUEUE ENTRIES (ANABOLISM)

SWAPQ ENTRIES (SWAPPER)

These areas are generally referred to as message areas. They are gotten from and returned to a set of spaces in memory called the message pool. The message pool is made of save spaces and structured as a two dimensional array (Figure 7-9) with the rowsize an MCP define - ROESIZE. This is the maximum area size that may be gotten. At halt/load time the pool is set up with the first four rows allocated.

Whenever a request is made to get or forget a message area, the two procedures GETAREA and FORGETAREA will try to maintain three completely empty rows. They do this by requesting another row whenever they exhaust one and by requesting that a row be forgotten when they empty it. The request for another row is done by causing the event the independent runner AREAMANAGER is waiting on.

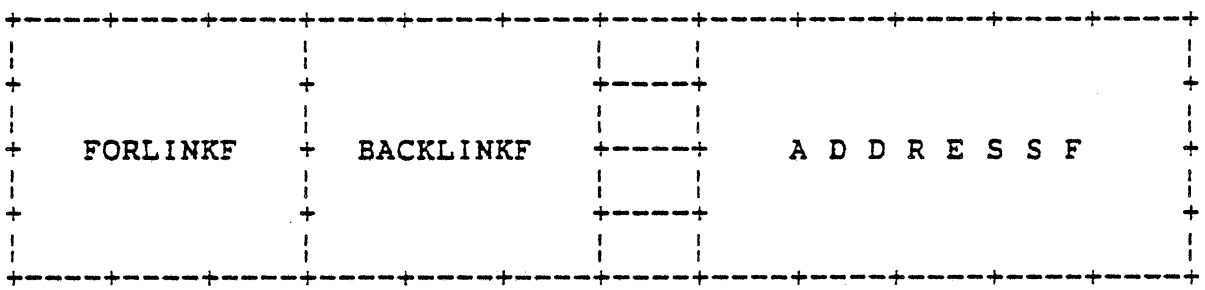
Within the rows, area management is kept very simple. Two links are used to specify the size of the available area and point past any in use areas (see figure 7-10). Areas are linked in size order. The head of the chain is the GETAREAHEADWORD and the tail is the GETAREATAILWORD. The links are protected by a lock called GETFORGET.



MSGVECTORS (IN MCPSTACK) IS A DOPE VECTOR THAT POINTS TO AREA POOL.

Figure 7-9. MSGVECTORS

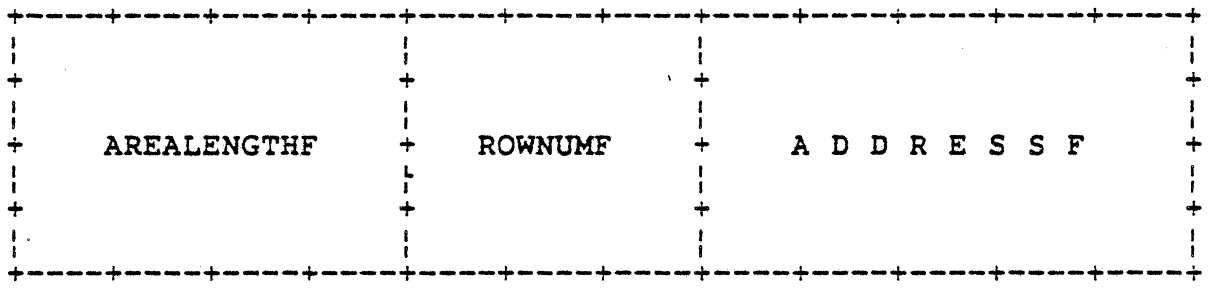
G E T A V A I L A T A G 5



[47:12] FORLINKF
[35:12] BACKLINKF
[19:20] ADDRESSF

FORLINKF ROW RELATIVE ADDRESS TO AREA IN FRONT OF THIS AREA IN THIS ROW.
BACKLINKF ROW RELATIVE ADDRESS TO AREA BEHIND THIS AREA IN THIS ROW.
ADDRESSF BACKLINK TO NEXT SMALLER AREA. THIS ADDRESS WILL SPAN ROWS. IT POINTS TO GETAVAILB WORD.

G E T A V A I L B T A G 5



[47:16] AREALENGTHF
[31:12] ROWNUMF
[19:20] ADDRESSF

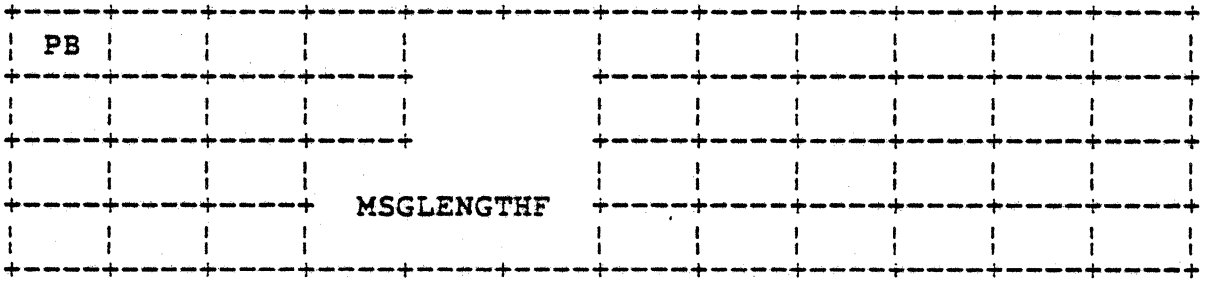
AREALENGTHF LENGTH OF AREA. SIZE INCLUDES LINKS.
ROWNUMF THE ROW NUMBER THIS AREA IS IN.
ADDRESSF LINK TO NEXT LARGER AREA. THIS LINK SPANS ROWS. THE LINK POINTS TO A GETAVAILB WORD.

Figure 7-10. Area Links (1 of 2)

THE AREA DESCRIBED BY THE LINKS IS AFTER THE GETAVAILB LINK WORD. THE LINKING IS THRU THE GETAVAILB LINK WORD. A LINK LIST LOOKUP OPERATOR IS USED TO SEARCH LINKS.

IN USE AREAS HAVE ONE LINK WORD. THE FOLLOWING IS A LAYOUT OF THAT LINK WORD.

I N U S E L I N K T A G 7



[47:1] PBITF
[33:10] MSGLENGTHF

PBITF IF ON THERE IS A DELTA (SLOP) WORD AT THE END OF THIS AREA.
MSGLENGTHF THE SIZE OF THE AREA. INCLUDES THE LINK.

Figure 7-10. Area Links (2 of 2)

WSSHERRIFF

WSSHERRIFF is the major procedure in the implementation of working set memory management. WSSHERRIFF has two jobs to do and these relate to two parameters of the working set system. These are:

OLAYGOAL

AVAILMIN

OLAYGOAL specifies what percent of in use memory is to be overlayed every minute. AVAILMIN specifies what percent of the total system memory should be kept available.

When OLAYGOAL is equal to 0, the WSSHERRIFF does not exist and working set is not active. If OLAYGOAL is greater than 0 then the WSSHERRIFF is an independent runner and will remain in the mix as long as OLAYGOAL remains greater than 0. There will be one copy of WSSHERRIFF for each box (tightly coupled) in the system.

WSSHERIFF will periodically go through and perform both of its tasks and then go back to sleep until the next interval. These intervals occur every 3 seconds.

WSSHERRIFF's first task is to make sure that the olaygoal specification is met. It does this by checking each stack's olaygoal. These are calculated in initiate and are less than the system olaygoal for high priority tasks, equal for priority 50 tasks and greater for lower priority tasks.

For each stack, this olaygoal is multiplied by its core-in-use and by a factor about like:

(# SECONDS SINCE LAST WSSHERRIFF DIV 60)

The result of this calculation will be the amount of in-use memory associated with this task that will be overlayed. This number is saved in OLAYCNTL for each stack.

After this number (called OLAYAMOUNT) is calculated for each stack the WSSHERRIFF starts searching through memory, overlaying overlayable areas according to each stack's olayamount. It starts this search at the LEFT-OFF pointer in MEMORY[MEMBASE+1].

For each memory area that WSSHERRIFF encounters it will find out which stack that the area belongs too and overlay it if OLAYAMOUNT is still positive. It will then decrement OLAYAMOUNT and continue the search. When it gets to the bottom of memory it will start over again at the top. Bottom

of memory as used here means MCPBASE plus DOSIZE (DOSIZE is the value found in SIZEF of MEMORY[MEMBASE]). See figure 7-8.

After this process is complete, the OLAYGOAL specification will be met.

Next, WSSHERRIFF will check AVAILMIN parameter. If the actual available minimum memory on the system is less than the amount specified by the parameter WSSHERRIFF will stop tasks and force them to have a temporarily high OLAYGOAL. If tasks are suspended but AVAILMIN requirement is met then WSSHERRIFF will restart them and return their OLAYGOALS to normal. Note that the system may pulsate if AVAILMIN is too high for OLAYGOAL is too low, and thrashing will occur if AVAILMIN is too low.

WSSHERRIFF AND MEMORY LINKS

As previously mentioned, WSSHERRIFF starts at the top of memory looking for overlayable in use areas and works his way down through memory until the base MCP area is hit. At this point WSSHERRIFF starts at the top again. The top as specified by this procedure is placed in the left-off pointer of MEMORY[MEMBASE+1]. This will place in the left-off field, the absolute address of the word shown in figure 7-11. Notice that the top four words in memory are "AVAIL" links that do not surround an area. The way WSSHERRIFF moves down through memory is as follows:

1. Where the left-off points minus 1 points will be a link word. It does not matter if this is an "AVAILZ" of "LINKZ" memory link. The size field in both types are the same. This field is subtracted from the left-off pointer and now the pointer is either at an "AVAILA" link or a "LINKA" memory link (point 1 in figure 7-11).

2. If the "INUSEF" in the word now pointed to by the left-off pointer is 1 then we are looking at a "LINKA" and must determine if the area should be overlayed. If it were assumed that the area at point 1 should not be overlayed then we go back to step 1. We conditionally go back to step 1 if the in use field is 0. It doesn't really make sense to overlay an available memory space.

Moving down through the links, WSSHERRIFF will eventually get to point 3 in figure 7-11. This point is the base of memory for the box. The pointer is set to the top of memory for the box again.

OVERLAYING

Overlaying occurs mostly, in instances where GETSPACE is trying to allocate a space in memory and can't find a space

large enough. In a case such as this, a few techniques still remain before we have to actually take a data array and write it to disk. One technique is quite simple and can be explained with the use of figure 7-12. Shown in this figure is an area in memory occupied by code. If it is assumed that the program using the 01 stack is working in segment 4, then we can forget space on segment 3 and use this area (possible the program won't even need its seg 3 code again).

Figure 7-11. Memory Links as Used by WSSHERRIFF

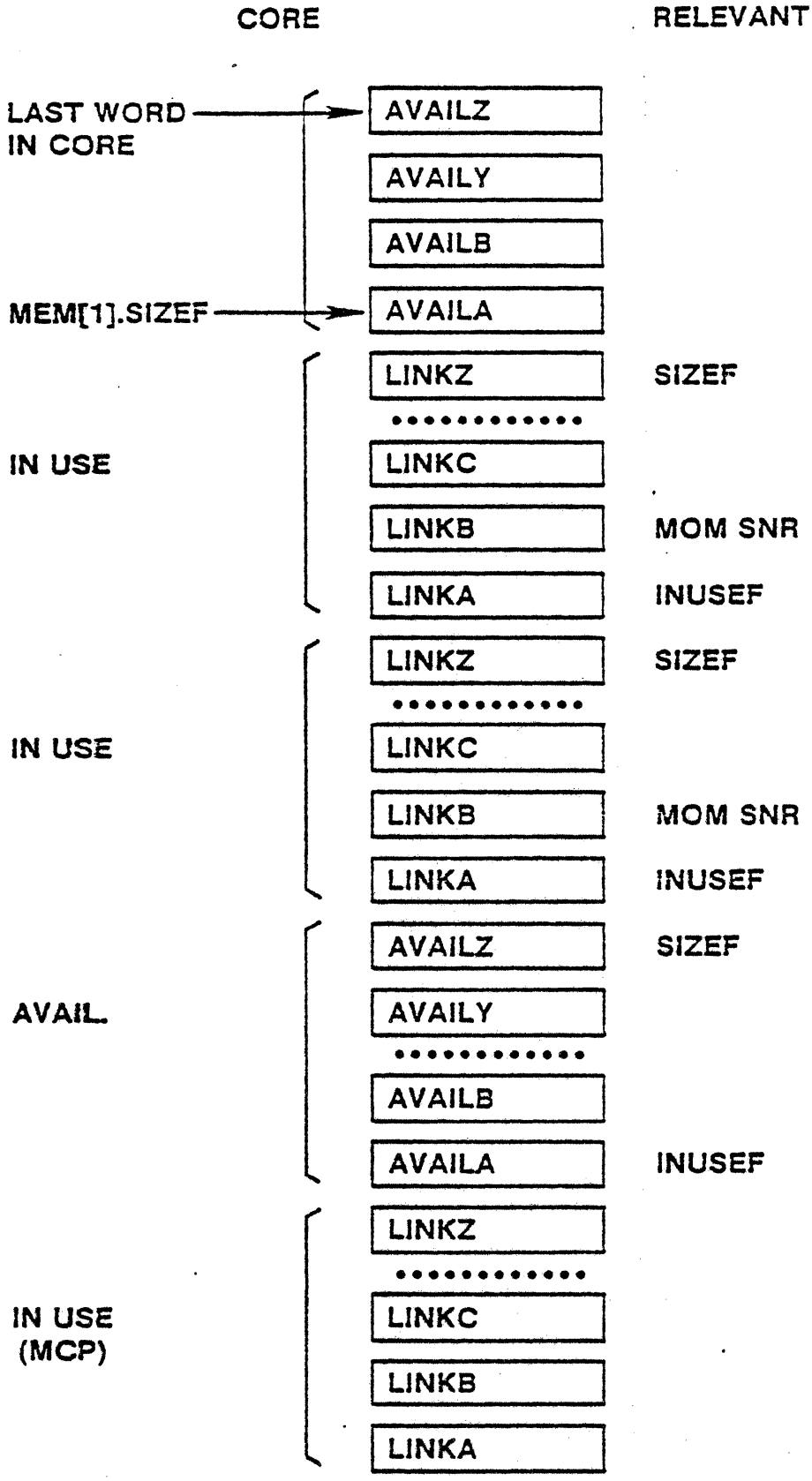
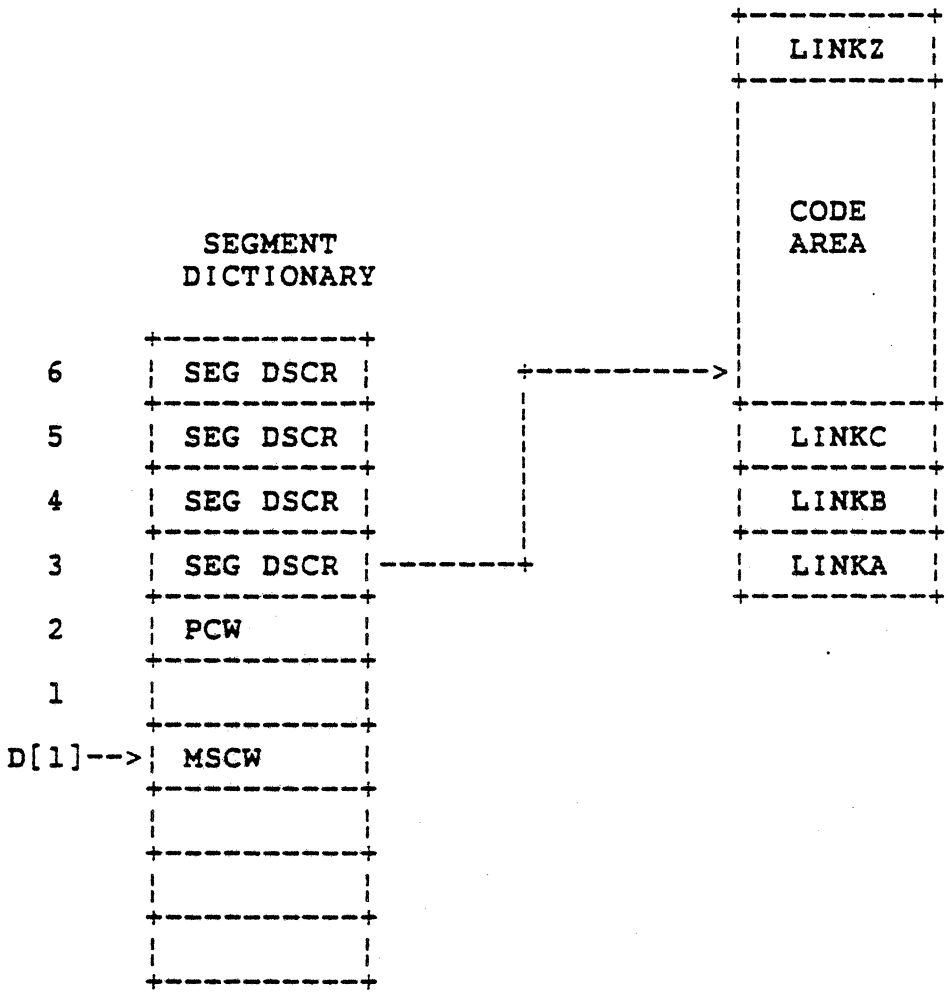


Figure 7-11 Memory Links as Used by WSSHERRIFF



CODEHEADERINDEX WORD OF PIB CONTAINS DISK HEADER INDEX OF CODE FILE.

Figure 7-12. Code Overlays

To forget space on a segment descriptor, all that is required is to take the disk address of the code (previously save in LINKB) and place it in the address field of the descriptor and turn bit 18 in the descriptor to indicate, to a possible later presence bit action, that the address code file relative. Of course, FORGETSPACE will place in the forgotten area, the "AVAIL" links and place the area in the appropriate avail lists.

In cases where a data array will have to be written to disk, use must be made of the program's overlay file. The overlay file is maintained by procedures FINDOLAYSPACE and LOSEOLAYSPACE which are used to get and return overlay disk for stacks. Users will get overlay space for data areas before they get the memory area. A user will be allocated as much

overlay disk as required. This file is fully described by two descriptors. Their names are:

OLAYINFODESC (Located in base of process stack)

OLAYFILEDESC (Located in PIB)

See figure 7-13. OLAYINFODESC, basically, points to a dope vector (the first word is the exception) and each descriptor in this vector is associated with an overlay row. Descriptor at index 1 is for the first row, descriptor at index 2 is for the second row, etc. Word 0 of the vector contains miscellaneous information and has the following format:

OLAYROWINUSE 47:1

EMPTYROWS 43:12

INUSEROWS 31:12

OLAYROWSIZE 19:20

OLAYROWSIZE comes from the value in the cold start "OLAYROW" card and is used to determine the number of segments in 1 overlay row and the number of words in 1 row of the OLAYINFODESC descriptors. The reason for this relationship is as follows: there is 1 bit required for each overlay segment to keep track of in-use and available segments in the rows. These bits are kept in the OLAYINFODESC array. If, in figure 7-13, word OLAYINFODESC [1,0] were brought to the top of the stack, then the top of stack bit 47 would be on if olay row 0, segment 0 was in use (a data array has been written to this segment). If this segment were available then bit 47 would have been off.

Once an available location has been found (olay file relative), the physical location of the row must be obtained. This disk location is described by a header (in standard format) that is not in the disk file header stack but is instead, pointed to by OLAYFILEDESC this word contains mostly zeros. Word 5, the FILESTRUCTURE word, is the only word in the fixed portion of the header that is used. The other words used are row address words, RAW, and contain the EU number and segment (in hex) of where the row starts. Disk overlay rows are obtained, when required, by GETUSERDISK, via the procedure OLAYSCOUT.

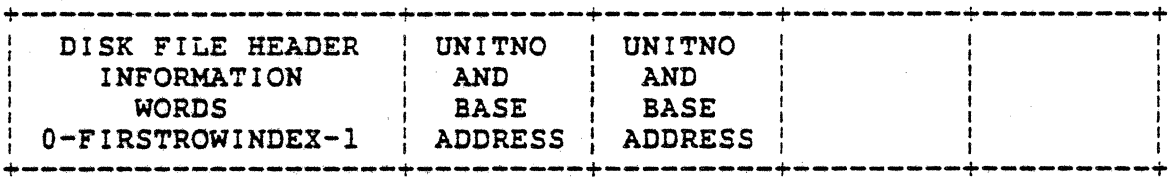
NEEDAROW

NEEDAROW is an MCP global array whose length is calculated as follows:

MAXSTACKS DIV 32 + 1

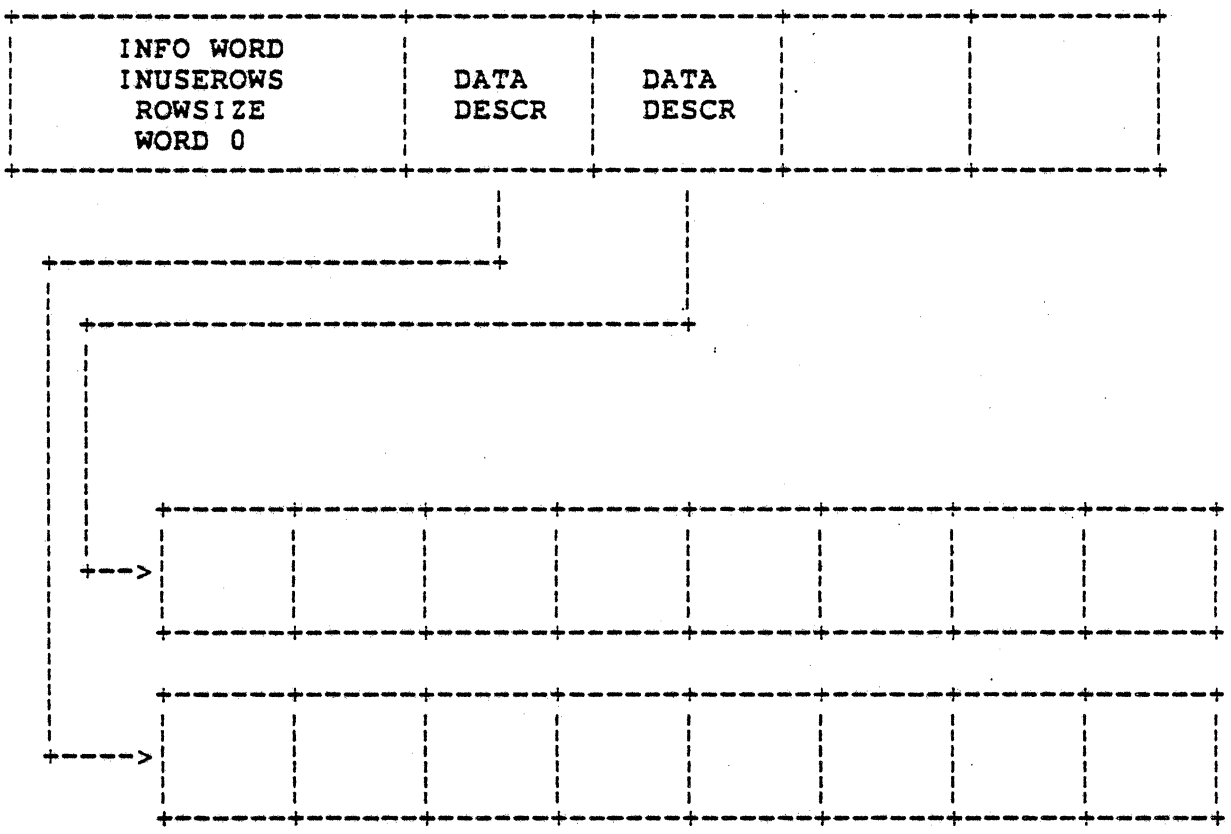
The basic procedure used to find overlay space in a given overlay file is named FINDOLAYSPACE. If this procedure can find no available areas left in the program's overlay file then a new row must be obtained when a new row is required, a bit (associated by stack number) is turned on and OLAYSCOUT is forked. OLAYSCOUT searches the NEEDAROW array looking for stacks that need overlay rows. By use of GETUSERDISK, OLAYSCOUT will get overlay rows and set up the corresponding RAW's. OLAYSCOUT terminates when no more olay rows are required.

OLAYFILEDESC - DISK HEADER FOR OVERLAY FILE. POINTED TO BY A WORD IN THE PIB (OLAYFILEDESC).



EACH BASE ADDRESS POINTS TO THE BASE OF OLAYROWSIZE SEGMENTS.

OLAYINFODESC - ALLOCATION MASK FOR THE DISK AREAS. POINTED TO BY A WORD IN THE STACK (OLAYINFODESC).



ONE BIT IN THE BIT MASK REPRESENTS ONE SEGMENT IN OVERLAY FILE. IF THE BIT IS ON THE SEGMENT IS IN USE. THE SIZE OF THE BIT MASK IN WORDS IS: (OLAYROWSIZE+47) DIV 48. ALLOCATION OF SEGMENTS IN THE OLAYFILEDESC IS CONTROLLED BY THE OLAYINFODESC. THERE WILL BE A BIT MASK DESCRIPTOR AND ROW IN THE OLAYINFODESC FOR EACH ROW ADDRESS WORD IN THE OLAYFILEDESC.

Figure 7-13. Overlay File Structure

PRESENCEBIT

PRESENCEBIT is passed the P2 parameter from HARDWAREINTERRUPT. This parameter may be one of the following:

1. Data descriptor.

2. Segment descriptor.

It is also passed a parameter which can be:

1. The RCW where the presence-bit occurred.

2. Information on how the P-bit is to be done. This is used for direct MCP calls.

The address field in data descriptors will be used in one of four different ways depending on the decoding of bits 17, 18 and 19 as follows:

BITS 19:3	HOW ADDRESS IS USED
0 0 0	Address is relative to the beginning of the ARRAY INFORMATION TABLE.
0 0 1	Address is relative to the beginning of the OWN ARRAY TABLE.
0 1 X	Address is relative to the beginning of a code file and will usually point to a value array or data pool.
1 0 X	Relative to the beginning of the stack's overlay file.

Note:

The X's above are part of the relative address. The address field (all 20 bits) in segment descriptors is always relative to the base of a code file.

PRESENCEBIT finishes by returning the present copy descriptor with the correct memory address back to HARDWAREINTERRUPT. A description of the general flow of PRESENCEBIT follows.

Outline of PRESENCEBIT

1. Check to see if Mom is locked. A locked mom will have all bits on in the address field. If tthe mom is locked, PRESENCEBIT will wait until it is unlocked.

2. The mom is checked to see if it is present. If so a branch is made to SAUL (Search And UnLock).

3. The mom is locked.

4. The stack number that has the mom is computed. This is found by searching from the mom down thru memory for a tag 7 word (PCW or LINKB memory link).

5. The length of the memory area required is computed. This is based on the length field and size field in the descriptor.

6. If the descriptor is segmented, a SEGDOPE only will be set up.

7. If the mom has a 4"00001" in the address field, the request will be for save memory.

8. If the area is not save, a check is made to be sure it will fit in one overlay row.

9. If the area is to be overlayable and no overlay disk has been allocated, it is requested. The number of sectors required is based on the size of the memory area. Descriptors which have a size field of 0 will allocated space for the tags (tag transfer will be done).

10. GETSPACE is called to get the memory.

11. If this is a segmented array, the dope vector is set up. Go to SAUL.

12. If this is first time allocation of an array, the area is zeroed. Go to SAUL.

13. If the mom points to data in the overlay file (bit 19 is on) the overlay file will be used to get the data.

14. If the mom points to data in the code file (bit 18 is on) the code file will be used. The proper code file must be used. A search is made to find the D1 stack in the environment of the mom.

15. The I/O is done to read the data.

16. Label SAUL. This stack is searched for copy descriptors. The copies are updated to point to the area.

17. The mom is updated and unlocked.

SECTION 8

DISK MANAGEMENT

INTRODUCTION

Disk, whether it be head per track or pack, is the major storage device on the large systems. As with memory, disk requires management of its areas. How these areas are managed, their size, their locations and content, etc. are of primary interest.

Figure 8-1 was drawn as in interpretation of a reading of the "B7000 System Miscellanea" manual and is accurate enough from a big picture point of view. What was called the LABEL BLOCK in this manual is actually the first 23 segments on the halt/load disk unit and is composed not only of the label, but also of the disk bootstrap (first segment of the 23). This bootstrap as well as everything else shown in figure 8-1 was created by the SYSTEM/LOADER.

When the loader is run for the purpose of cold starting, it has two important tasks:

1. Load the MCP from tape to disk (by default, the first row of the MCP's code file will start at segment 25000).

2. Set up the FLAT DIRECTORY and LABEL AREA.

HALT/LOAD DISK LAYOUT

Figure 8-2 shows the layout of the halt/load disk as set up by the SYSTEM/LOADER. In absolute segment 0 is the boot, followed by the label in segment 4. Most of the first 28 segments are not used but are reserved. By default, the FLAT DIRECTORY starts at segment 28 (This is a define on ""PKMINADDR"") and this segment is what is pointed to by the arrow leaving the LABEL BLOCK of figure 8-1.

TERMINOLOGY

The disk sub-system on large systems contains head-per-track disk or disk packs or any combination of the two. We will consider as as disk unit each part of the system which can be considered as an area of contiguous disk for addressing purposes. For head-per-track disk this will be a data memory bank (the set of storage units controlled by one electronics unit) and for exchangeable disk it will be a disk pack.

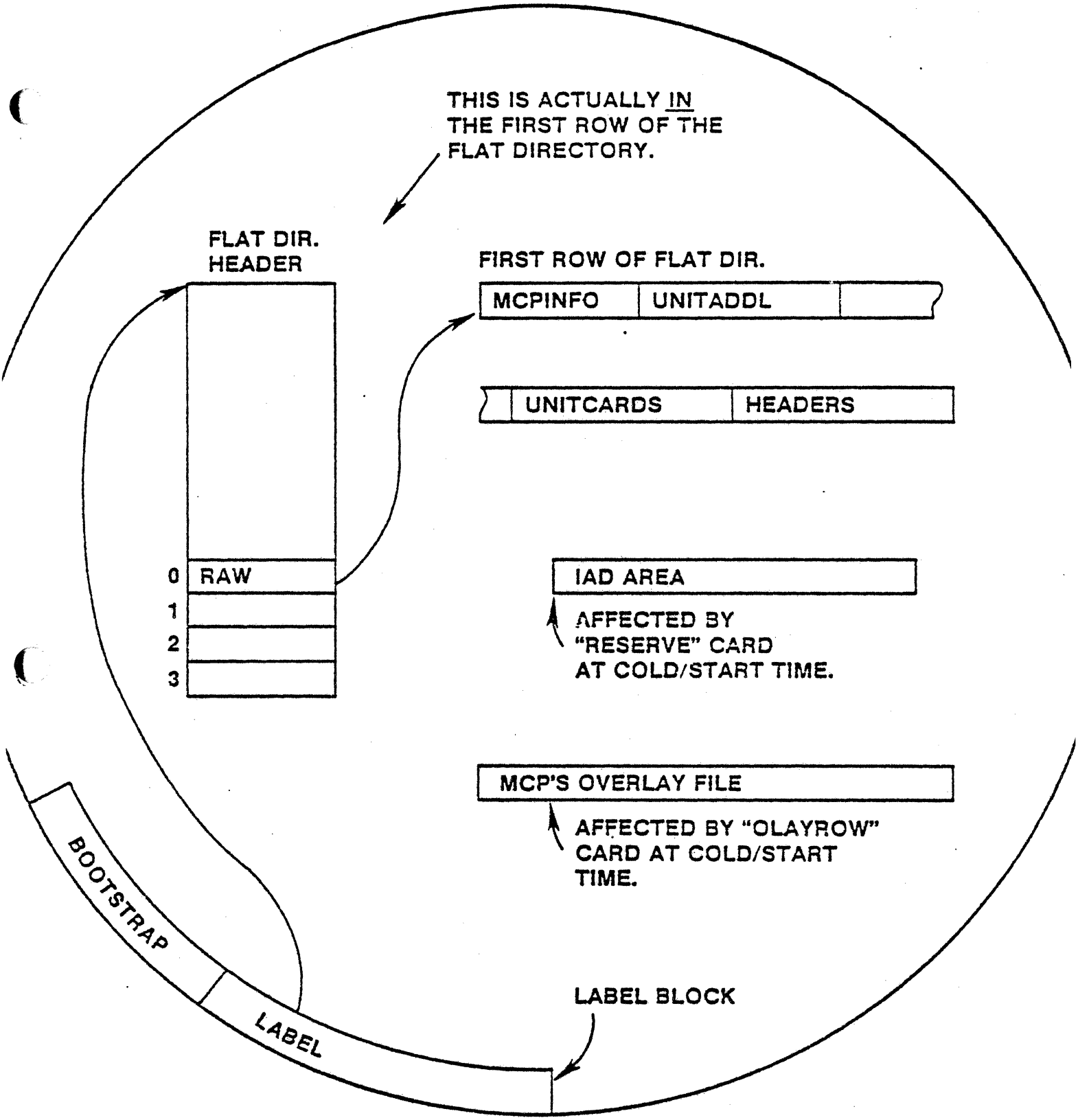


Figure 8-1. Disk Initialization

Figure 8-1. Disk Initialization

For allocation of storage space for files, we specify on which unit(s) a file may reside by specifying a FAMILY NAME. The name refers to a disk family, a unit or group of units which has its own directory stored on one unit of the family. Thus, when accessing a file, the file name and family name must be given, e.g., FILE ON PAKFAM1, A/B ON DISK. (if no family name is given the default of disk is assumed).

The units within each family are assigned a FAMILYINDEX, (counting up from 1 for all units in the family). In the case of packs the unit need not be present on the system if a file is required which is on a unit not currently on the system, the operator will be requested to load that unit.

The directory for the system consists of two parts: the directories for each family, known as FLAT DIRECTORIES, and the ACCESS STRUCTURE. The FLAT DIRECTORY contains HEADER-NAME BLOCKS for each file in the family. These blocks contain all information about the files they are associated with. These HEADER-NAME BLOCKS are in an arbitrary order, which is not very useful when trying to locate a file by name. Thus, to find a HEADER-NAME BLOCK given the file name (and family name), the ACCESS STRUCTURE is used.

Data integrity is maintained in the FLAT DIRECTORIES and ACCESS STRUCTURE by means of CHECKSUM words in each block. Thus, before a block is used, it can be checked to ensure that it has not been overwritten or corrupted in some manner.

FLAT DIRECTORIES

For each family there is a FLAT DIRECTORY which contains HEADER-NAME BLOCKS for each file in the family. The unit that contains the primary copy of the FLAT DIRECTORY is known as the DIRECTORY UNIT. Up to two copies of this directory (three total) can be maintained on other members of the unit's family.

The FLAT DIRECTORY has 600-segment rows, the first row of which, is shown in figure 8-3. All rows in a FLAT DIRECTORY other than the first row have a different format and mostly contain headers with only one segment reserved for MCP use.

The HEADER-NAME BLOCKS (figure 8-4) of FLAT DIRECTORIES contain the name of the file and information about the file, including security information, record block size, file type, and address of each row of the file (row address words).

In the first row of the FLAT DIRECTORY (figure 8-3) the first 142 segments contains entries in fixed locations. Of particular importance is the MCPINFO table which is stored in segments 5 through 14. This table is created by the SYSTEM/LOADER and maintained by the MCP. Its basic function is to hold information that should be remembered across

halt/loads. Information contained in these segments is the last mix number used, name of the intrinsics file, number of auto printers going, etc. Segments 24 through 59 contain the header for the FLAT DIRECTORY of which it is a part (sometimes referred to as a self-pointer). In the section referred to as ""B7700TABLES"" is where the parameter card information concerning peripherals and DCP's is kept.

A minimum of two disk accesses is required to actually bring in a header from the FLAT DIRECTORY. One access is required to read in the appropriate section of the ACCESS STRUCTURE and with use of this information, the header in the FLAT DIRECTORY may be read into memory.

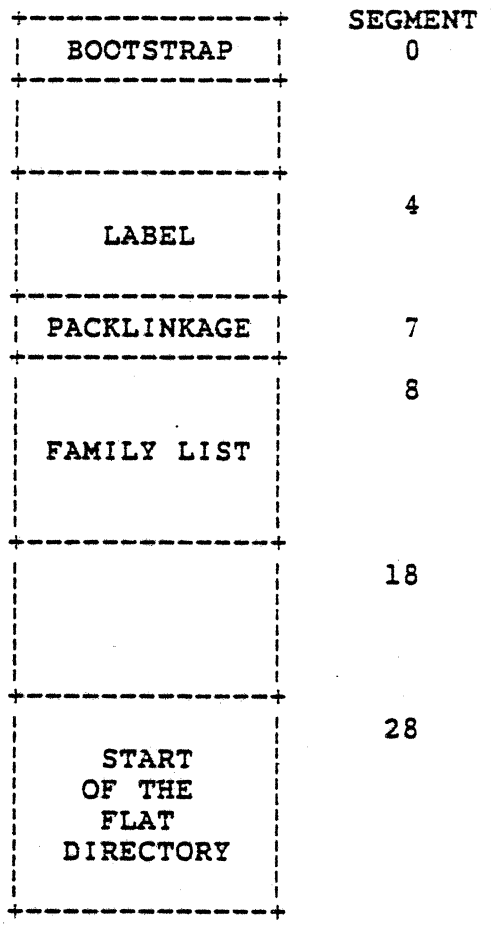


Figure 8-2. Halt/Load Disk Layout

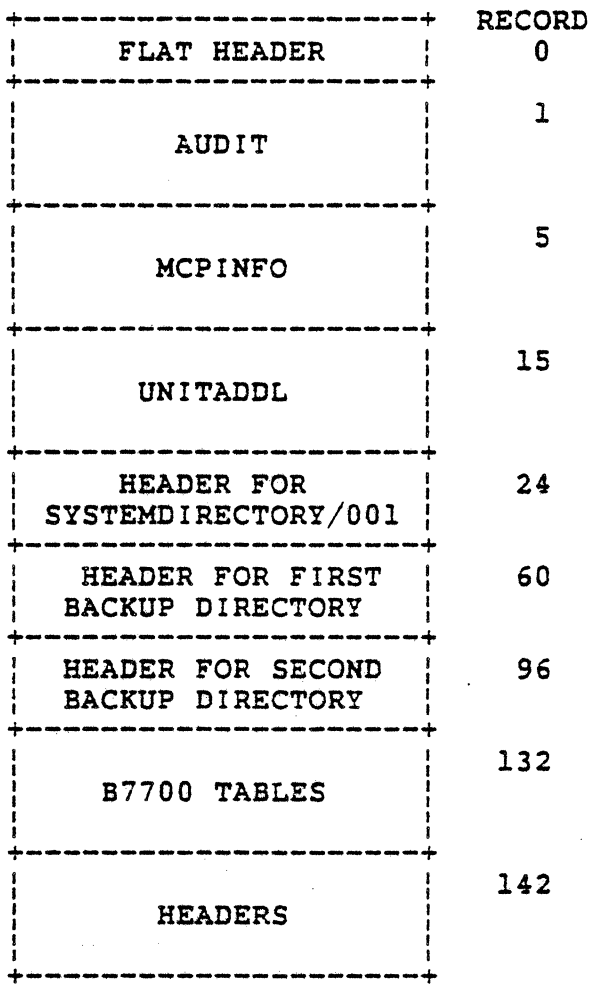
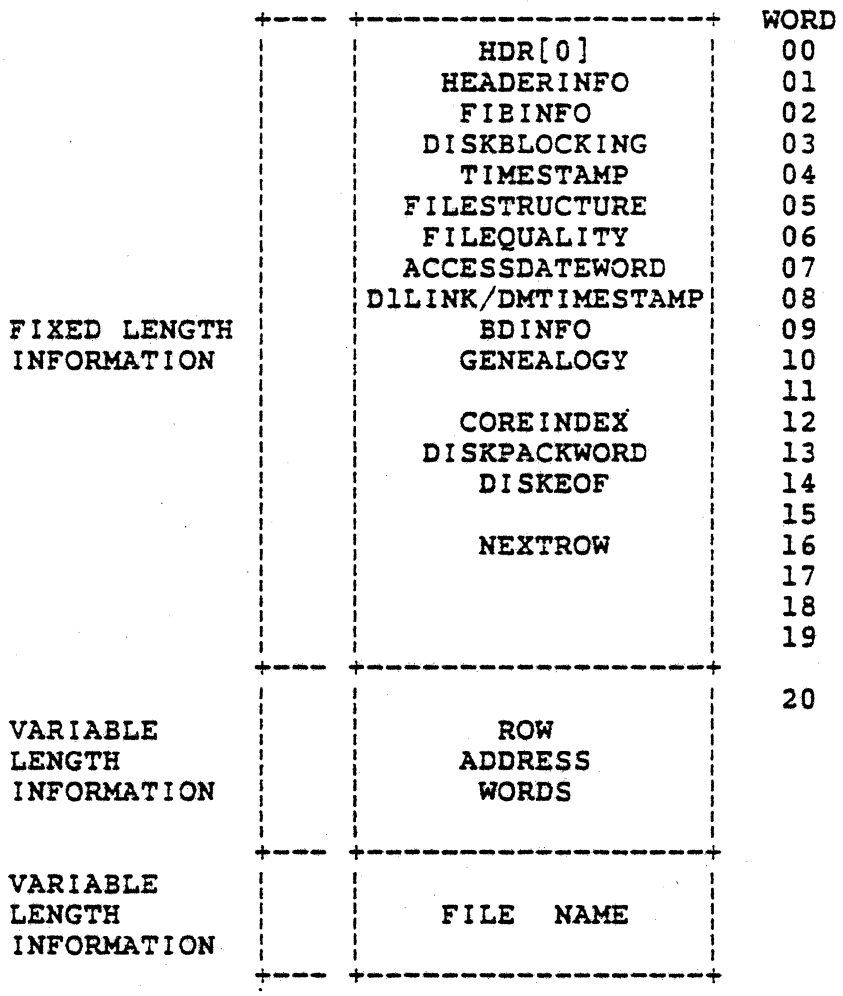


Figure 8-3. First Row of the Flat Directory



ROW ADDRESS WORDS POINT TO ROWS OF FILE NAME POINTED TO BY HEADERINFO WORD

Figure 8-4. Header-name Block

ACCESS STRUCTURE

The ACCESS STRUCTURE is the means by which a file name may be used to find information about a file. There is one ACCESS STRUCTURE for the system and given a family name and file name within the family, it is used to produce the position of the file HEADER-NAME BLOCK within the FLAT DIRECTORY for the family (see figure 8-5).

The ACCESS STRUCTURE is constructed at system initialization time by the MCP (not the loader) from the FLAT DIRECTORIES of all families on the system at that time. If a new family is added to the system the ACCESS STRUCTURE will be enlarged to include the files in the new family. When a family is removed from the system, however, its file's entries remain in the ACCESS STRUCTURE. If at any time the ACCESS STRUCTURE is found to be corrupt (CHECKSUM errors, entries don't match FLAT DIRECTORY) it will automatically be recreated without requiring a halt/load.

There are two forms of ACCESS STRUCTURE. The CATALOG version contains information about all available versions of a file, the NON-CATALOG version contains information about the files which are currently accessible to the system. We will consider only the NON-CATALOG version.

One of the families on the system is designated as the one on which the ACCESS STRUCTURE is stored. This family, known as the CATALOG family, can be defined by the DL ODT command, otherwise it will default to the halt/load family (the one on which the running MCP resides).

The ACCESS STRUCTURE is stored as the file SYSTEM/ACCESS/NNN, where NNN is the FAMILYINDEX of the unit on which it is stored, and is structured in 8-segment blocks.

The ACCESS STRUCTURE is in two parts (see figure 8-6), the PACK ACCESS STRUCTURE, PAST, and the FILE ACCESS STRUCTURE, FAST. The PAST is used to indicate which part of the FAST contains the entries for a given family name and the FAST blocks, when located, are used to produce a pointer into the FLAT DIRECTORY, given a file name. When you hear the word "PAST" you should immediately think "FAMILY" and when you hear the word "FAST" you should immediately think "FILE".

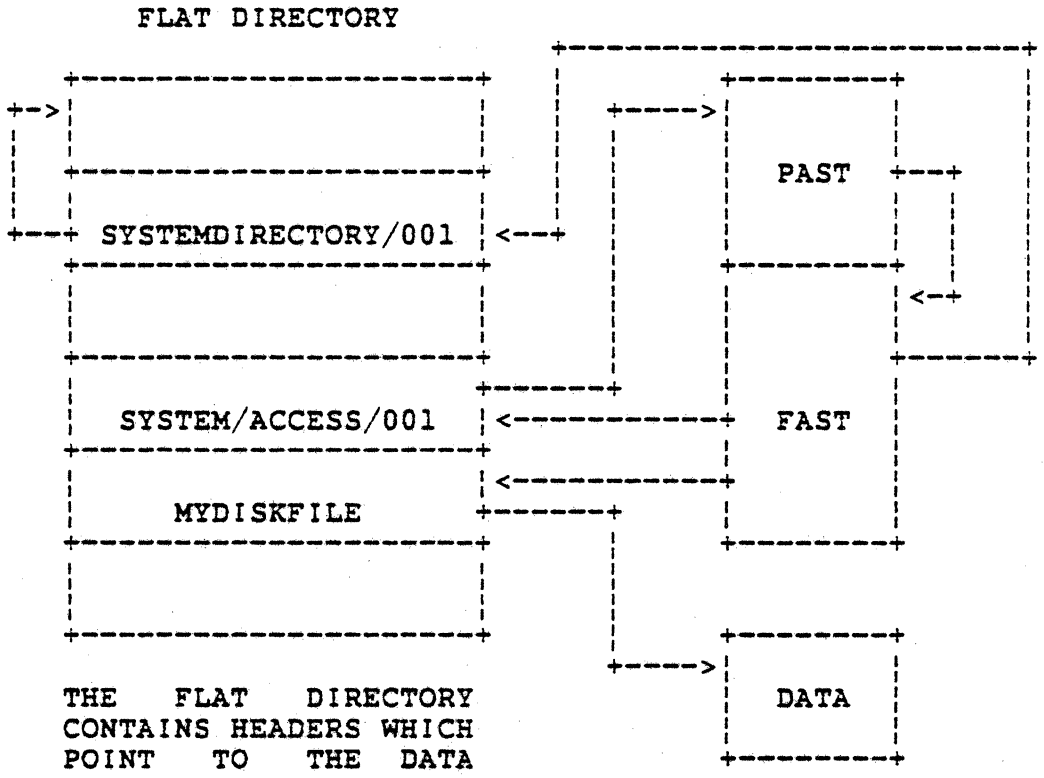
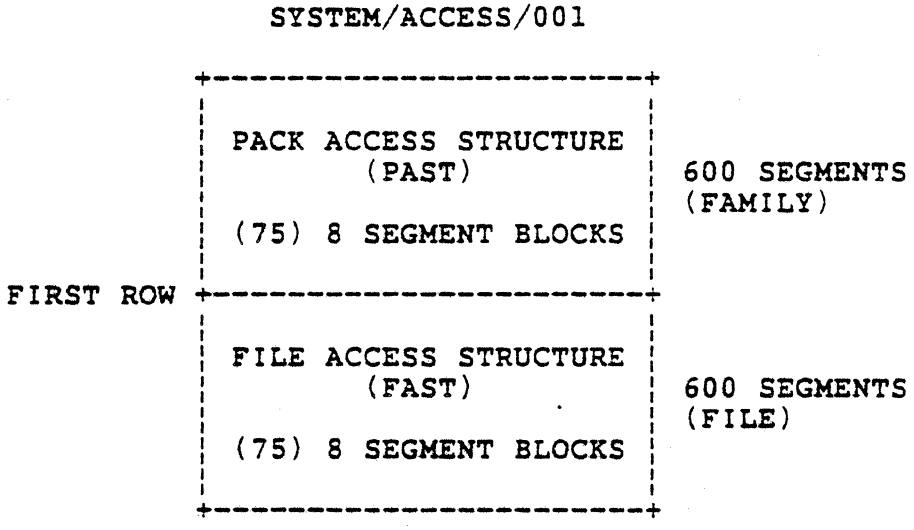


Figure 8-5. Disk Files



BUILDHEADER BUILDS THE HEADER "SYSTEM/ACCESS."

INITIALIZECATALOG SETS UP THE BASIC STRUCTURE.

CREATEFAMILY MAKES ENTRIES IN THE PAST.

PASTSEARCH LOOKS FOR ENTRIES IN THE PAST.

FILEHANDLER MAKES AND LOOKS FOR ENTRIES IN THE FAST.

Figure 8-6. Access Structure

PACK ACCESS STRUCTURE (PAST)

The PAST has a fixed size of 75 blocks, each of 8 disk segments. To find an entry in the past for the family, a block is calculated based on an arithmetic manipulation of the characters in the family name (ie. HASHED). This entry contains pointers identifying the section of the FAST which is applicable to the family. A simple search is then performed on the PAST block until the entry with the given family name is found. Given below is an example past entry. Note that all PAST, as well as FAST, blocks are character oriented (1440 characters per block).

CHARACTER	CONTENT	COMMENT
00	01	PNUM-NUMBER OF FAMILY MEMBERS.
01	04	
02	C4	PNAME- FAMILY NAME. FIRST
03	C9	CHARACTER IS THE NUMBER OF
04	E2	CHARACTERS IN THE NAME.
05	D2	
06	00	PSERIAL- SERIAL NUMBER IF PACK
07	00	AND UNIT NUMBER IF HPT.
08	20	
09	18	PTIMESTAMP- TIME THIS ENTRY WAS
10	BB	MADE.
11	5B	
12	DB	
13	70	
14	34	
15	00	PFASTIX- BEGINNING FAST BLOCKS
16	09	FOR THIS FAMILY'S USERCODE AND
17	40	AND NON-USERCODE FILES
18	00	(BEGINNING OF FILE RELATIVE).
19	95	

The area designated as PFASTIX is actually composed of two fields. The first field is the first 5 hex digits (00094 in the example) and the remainder is the second field (00095). The 4""00094"" indicates that the starting FAST block for the ""DISK"" family's NON-USERCODE files is block 4""94"", beginning of file relative. The 4""00095"" indicates that the starting FAST block for the ""DISK"" family's USERCODE files is block 4""95"", beginning of file relative. This is the initial set up for all families and represents, what could be, the beginning of two separate strings of blocks, one string for NON-USERCODE files and the other string for USERCODE files.

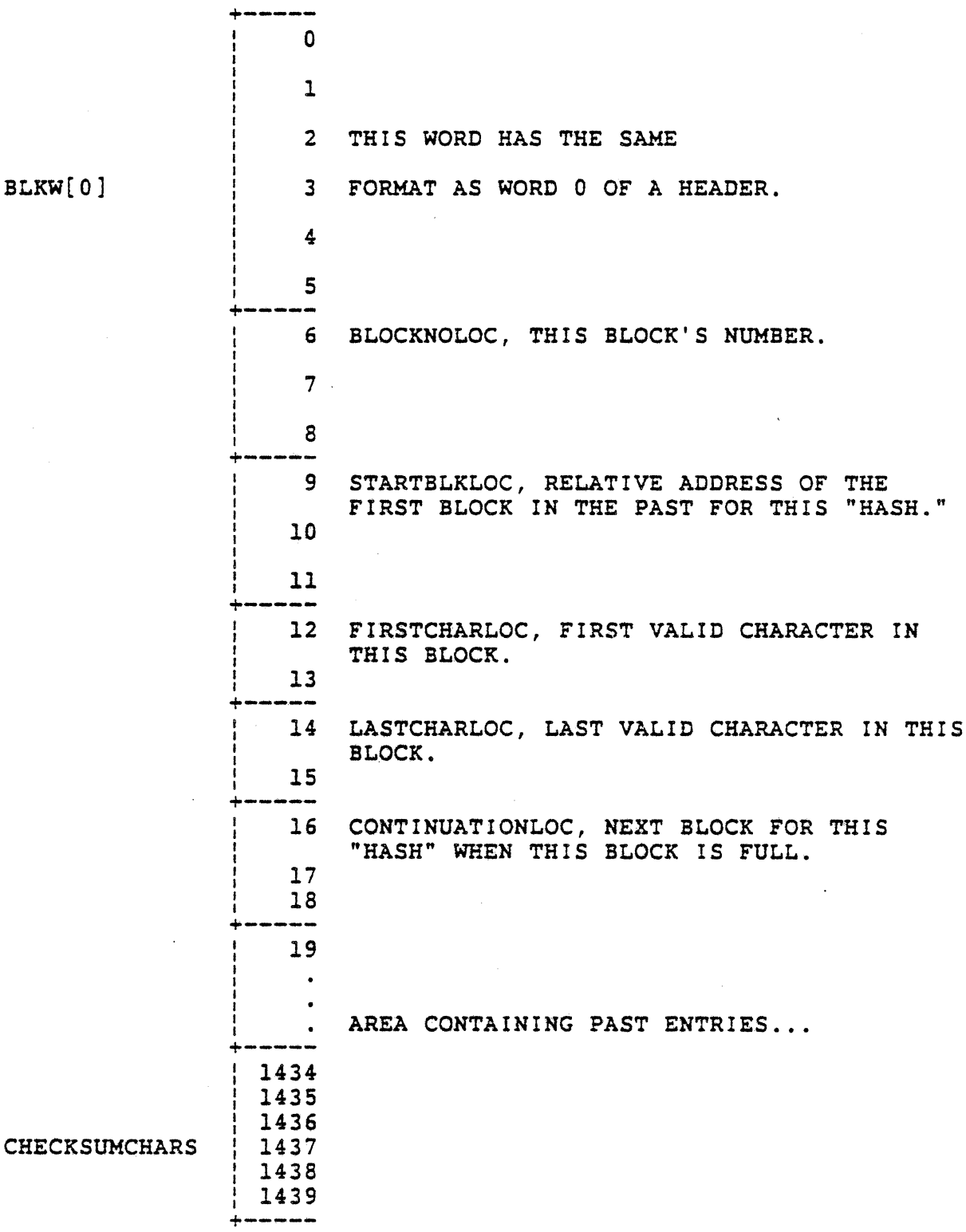


Figure 8-7. Pack Access Structure (Past) Block

When a family is brought on line, the MCP procedure READADISCLBL is forked. This procedure gets a memory area for an information array (UINF) and places information about the family in this array. When the label is read, the family name is obtained, and from this information the PAST is checked to see if the family's entry already exists in the ACCESS STRUCTURE. If it does, than all information from the PAST entry (refer to the example above) is stored in the UINFO array. It should be clear by now, that after the initial process of bringing the unit on line, no further ""PAST"" accesses are required. We can go directly from the UINFO array to the specified location in FAST and perform our search for the file name when needed.

For the structure of an entire PAST block (all are the same) see figure 8-7.

FAMILY NAME HASHING

A good example of family name hashing can be seen in the MCP procedure PASTSEARCH. Given a family name, two steps are taken to produce the PAST block number the family entry may be in (see figure 8-8). Assume the family named "DISK" is in the 3-word array BLKW. The for loop at the top of this figure is the first step and will be looped through three times. This produces RESULT 1. This result is now used in step 2 to produce RESULT 2. This is the final result and is the block number associated with the family name. Notice that for the family "DISK" the block number is 28.

```
FOR IX  := 0 STEP 1 UNTIL 2 DO
    HASH := (BOOLEAN(BLKW[IX]) EQV HASH).[44:48]:

                  4         3         2         1         0
BIT NO:   765432109876543210987654321098765432109876543210

BLKW[IW]  000001001100010011001001111000101101001000000000

HASH      000000000000000000000000000000000000000000000000

(EQV)     111110110011101100110110000111010010110111111111

.[44:48]  110110011101100110110000111010010110111111111111

BLKW[IX]  000000000000000000000000000000000000000000000000

HASH      110110011101100110110000111010010110111111111111

(EQV)     001001100010011001001111000101101001000000000000

.[44:48]  001100010011001001111000101101001000000000000001

BLK[IX]   000000000000000000000000000000000000000000000000

HASH      001100010011001001111000101101001000000000000001

(EQV)     110011101100110110000111010010110111111111111110

.[44:48]  011101100110110000111010010110111111111111110110 (RESULT 1)
```

(RESULT 2 - FINAL RESULT)

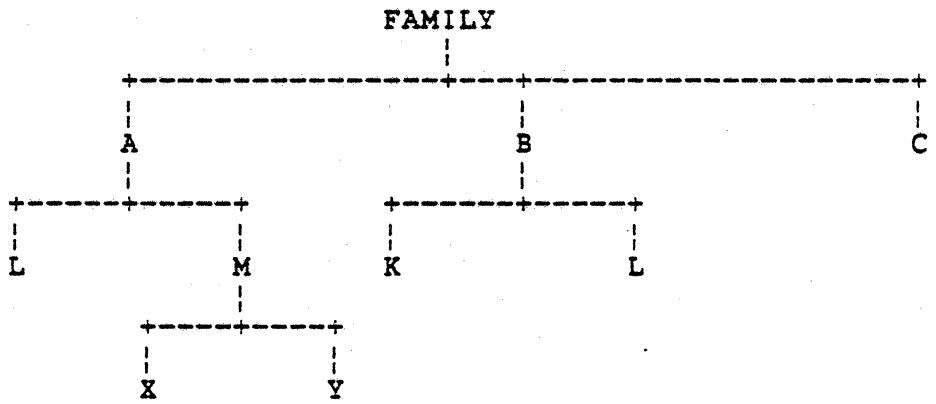
```
    BLKNO:= (REAL  (HASH).[37:38] + REAL (HASH).[47:10])
                  MOD PASTMOD + 1;     % PASTMOD = 75.

       HASH.[37:38] = 10110000111010010110111111111111110110
(PLUS) HASH.[47:10] =                                0111011001
                     ----------------------------------------
(TOTAL)              10110000111010010111000000000111001111
(MOD)                                                1001011
                     ----------------------------------------
(REMAINDER)                                            11011
(PLUS)                                                     1
                     ----------------------------------------
(TOTAL)  BLKNO                                         11100
```

Figure 8-8. Family Name Hashing

FILE ACCESS STRUCTURE (FAST)

The FILE ACCESS STRUCTURE for each family can logically be considered as a tree structure. Thus, if a family had the files: A/L, A/M/X, A/M/Y, B/K, B/L and C, the logical structure would be:



In the directory system that was previously in operation on the large systems, the directory was physically tree structured in this way, with separate disk areas allocated for each name. This was inefficient, both in terms of disk space and in the number of disk accesses required to find a file. In the post II.6 versions the information is held in a compact, character oriented form which minimizes both disk space requirements and number of accesses.

Each identifier in the tree has an entry associated with it in the FAST. Entries at the same level in the tree, and which have the same predecessors, are termed brothers, and are stored contiguously in order. (in the tree shown above, the identifier sets (A, B, C), (L, M), (X, Y)< and (K, L) are sets of brothers). Entries are termed FILES if the identifier is the last identifier of a file name (eg., X in A/M/X), and DIRECTORIES if there are further identifiers in the file name (eg., A and M in A/M/X). If an entry is a file entry it contains a pointer to the HEADER-NAME BLOCK. This pointer is a segment index relative to the beginning of that family's FLAT DIRECTORY. For entries that point to directories, the entry will contain a character index in the block to the next level of the tree, i.e., of the first son.

To demonstrate the tree structure we use the notation:

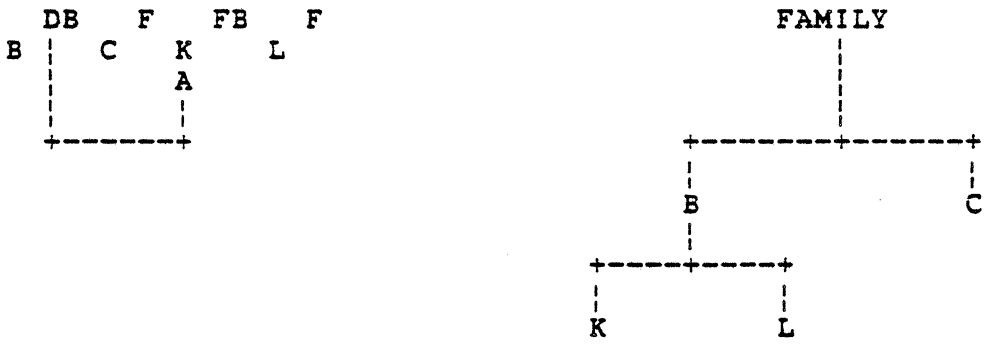
F for file.

D for directory.

B if the very next entry is a brother to this

entry.

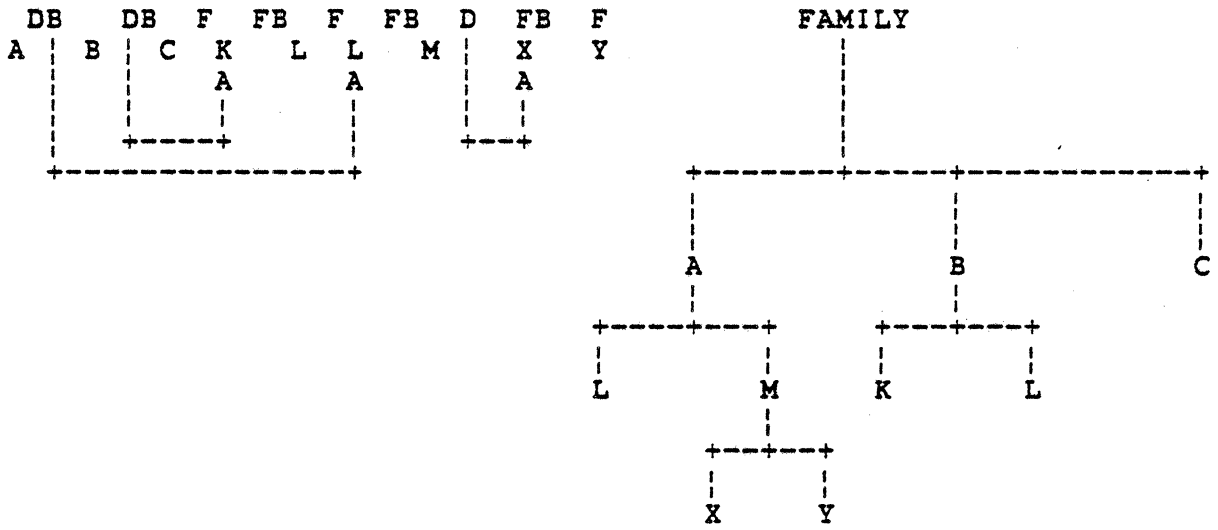
Suppose that the family described above started with the files B/L B/K and C. The physical layout of the FAST block for these entries would be:



The arrow in the FAST layout indicates the offset in characters to the next level (first son).

After the files A/L, A/M/X and A/M/Y have been added, the layout would be:

(CORRESPONDING TREE)



An example FAST entry for file C has been given below:

CHARACTER	CONTENT	COMMENT
00	27	POINTS TO A FILE AND HAS NO BROTHER.
01	01	THERE IS 1 CHARACTER IN THE FILE'S NAME.
02	C3	NAME OF THE FILE IS 8""C""
03	04	HEADER IS AT OFFSET 4""F00" IN
04	00	THE FLAT DIRECTORY AND IS 32
05	OF	WORDS LONG.
06	00	

The way the above information was obtained was by use of the information provided below:

Character 0 in a FAST entry is called the ""ID"" character and has the following format:

7:1 BROTHERF, on if this file has a brother.

6:2 IDTYPEF, broken down as follows:

6:1 DIR, on if this entry points to a directory entry.

5:1 FYLE, on if this entry points to a file.

NOTE:

IF BITS 5 AND 6 ARE BOTH OFF, THE LAST TWO CHARACTERS IN THE ENTRY CONTAIN THE ""CONTINUATION BLOCK"" THE ENTRY MAY BE FOUND IN.

4:5 ENTRYL, length of this entire entry in characters.

Character 1 in a FAST entry is the length of the name in characters.

Characters 2 through N are the name. N represents the last character in the name.

If the FYLE bit is on, the four characters following the name are in the format given below:

31:11 FILELENGTHF, length of the header in words.

20:21 FILELOCF, points to the header in this family's FLAT DIRECTORY and is given in segments relative to the beginning of the FLAT DIRECTORY.

If the FYLE bit and the DIR bit had also been on then the four characters described immediately above would have been followed by two characters of the following format (note that this is not the case in the example above):

> 15:16 DIRDELF, pointer to the next level and is given in characters relative to the beginning of the entry.

If the DIR bit had been on and the FYLE bit had not been on then the ""4"" characters described above would have not existed in the entry and in this specific entry, there would have been only 5 characters.

Figure 8-9 shows how an entire FAST block is layed out. Notice its similarity to the PAST block layout of figure 8-7.

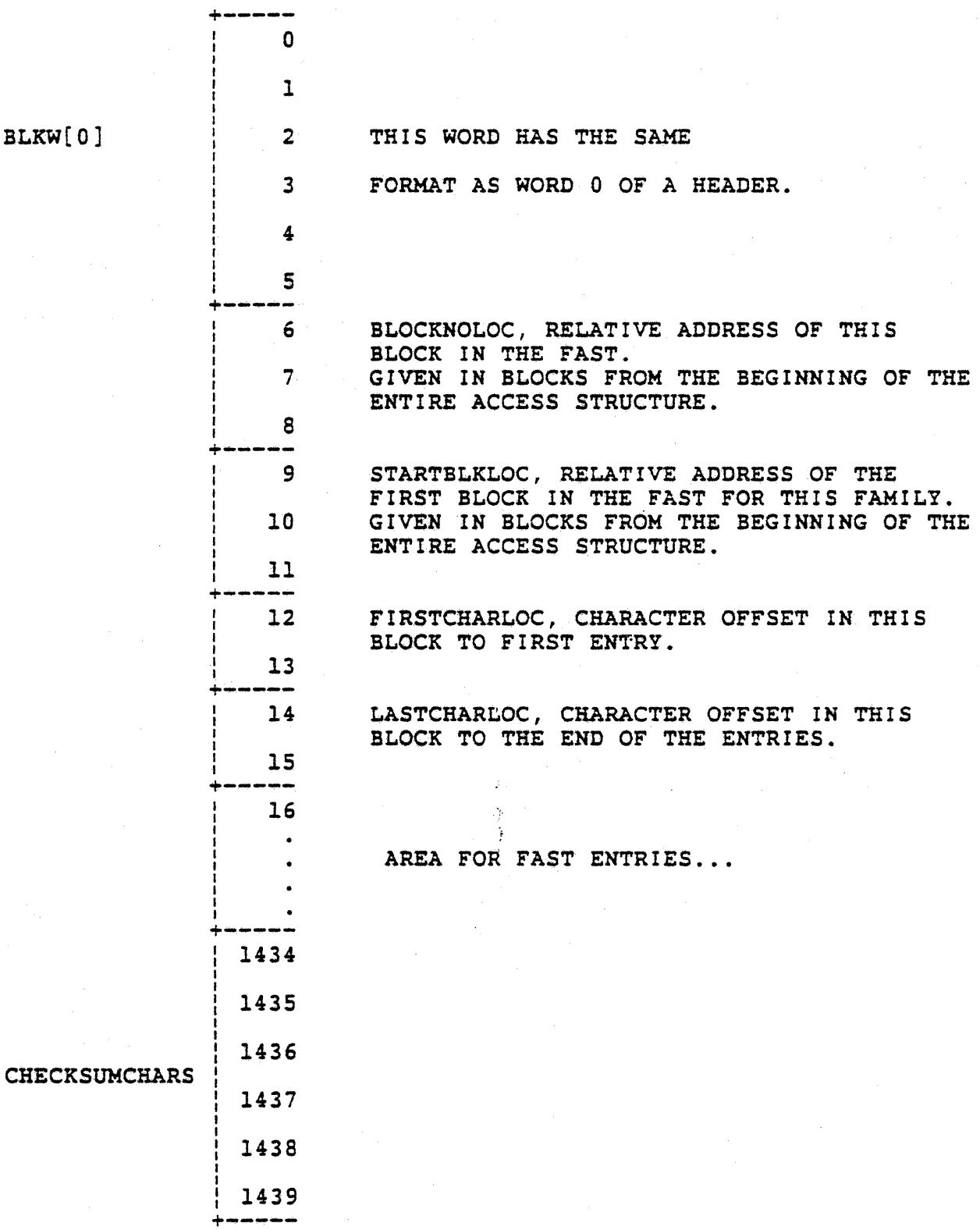


Figure 8-9. File Access Structure (Fast)

DIRECTORY COMPLEMENTING

At any given time an area of disk may be in one of three states. It may be available for future use by the system or it may be allocated to a file that has an entry in the disk directory (termed a permanent file) or it may be temporarily in use, either by the MCP or by a program which has opened and written information into the file but has not yet entered it into the disk directory.

When it is required to reinitialize the system (halt/load) the MCP needs to return all disk which was temporarily in use to the list of available disk. The easiest way to do this is to rebuild the list completely, and at the same time check the directory for any possible errors. The MCP can do this simply, because all disk that is not in use by files in the disk directory must be available at restart time. Thus it constructs the available disk list for each family by complementing the FLAT DIRECTORY, i.e., it starts with the whole disk area available and then, as it reads each header, it removes the disk allocated to the file from the available list.

The fact that directory complementing is done from the FLAT DIRECTORIES present on the system at restart time makes it a very fast process, since the large blocking factor of the FLAT DIRECTORIES enables them to be read with minimal I/O operations. The halt/load process, of which directory complementing is a part, is speeded up as a consequence of this.

GETUSERDISK

GETUSERDISK is the procedure called any time that disk space is required. This procedure gets space for one row each time it is called and will take one of two possible actions to look for the space if a family index is passed then space on that member only will be checked but if the base unit number is passed then every on-line member will be checked for the best possible fit. In this case, no preference is made to member or location of the area on a particular member. The family's base unit must always be passed but the family index is optional.

If GETUSERDISK can find enough space on disk, a row address word is returned and the area removed from the DISCAVAIL table. If the area had previously been allocated and the unit is now on-line, the PBITF will be on in the returned row address word. If GETUSERDISK could not find an area large enough, a word will be returned with the SIGNBITF on.

AVAILABLE DISK TABLES

AVAILABLE DISK TABLES are those tables used to keep track of available disk areas and available header locations in the FLAT DIRECTORY. These tables are located by descriptors in the BASE UNIT's UINFO array and are originally built by DISKMAPPER (and FLATREADER) during system initialization. Each family has its own set of these tables.

The table used for remembering available header locations is known as the FLATAVAIL table and may be seen in figure 8-10. The table used to keep track of the available user disk locations is called the DISCAVAIL table and is actually a set of tables (rough and fine tables). These tables may also be seen in figure 8-10 but have been redrawn in figure 8-11 to better display them.

The CHUNKLISTS (fine tables) of figure 8-11 are the last link to available user disk locations. All CHUNKLIST words have the following format:

> 47:24 DISKAREASIZEF, number of segments in area (in hex).
>
> 21:22 DISKAREAADDRF, absolute address of area (in hex).

When looking over the two fields described above, it should be noticed that no unit number or family index has been given. Realizing that a physical unit number must eventually be obtained we must look back toward the UINFO array for help. Our way out of this situation is quite simple. There is a relationship between the arrays GETHEAD and FORGETHEAD (the rough tables) and the CHUNKLISTS that allows us to get a family index and with this number, the FMLYLIST (figure 8-10) can be indexed and there is found a word with that members physical unit number.

The arrays GETHEAD and FORGETHEAD are the rough tables of available user disk. All words in both arrays have the following format:

> 47:16 USERDISKINDEXF, this is an index into the middle array (USERDISKLIST) of figure 8-11 and is used to relate the word GETHEAD or FORGETHEAD with a particular CHUNKLIST.
>
> 29:08 EUNOF, in this case the field's name is misleading. This field contains the family index of the member who owns the CHUNKLIST pointed to by bits 47:16 of this word. Two members should never be pointing to the same CHUNKLIST.
>
> 21:22 SEGADDRESSF, this field contains different

info depending on whether it is in a GETHEAD or FORGETHEAD word. In FORGETHEAD, this field contains the address of the lowest available area in the list pointed to by bits 47:16 but in GETHEAD this field contains the size in segments of the largest area in the CHUNKLIST pointed to by bits 47:16.

The GETHEAD array is used by GETUSERDISK. This array is set up such that all entries for a given family are grouped together with the largest size chunk closest to the end of the list. To illustrate this, a typical GETHEAD array is shown below:

Figure 8-10. Disk Information Arrays

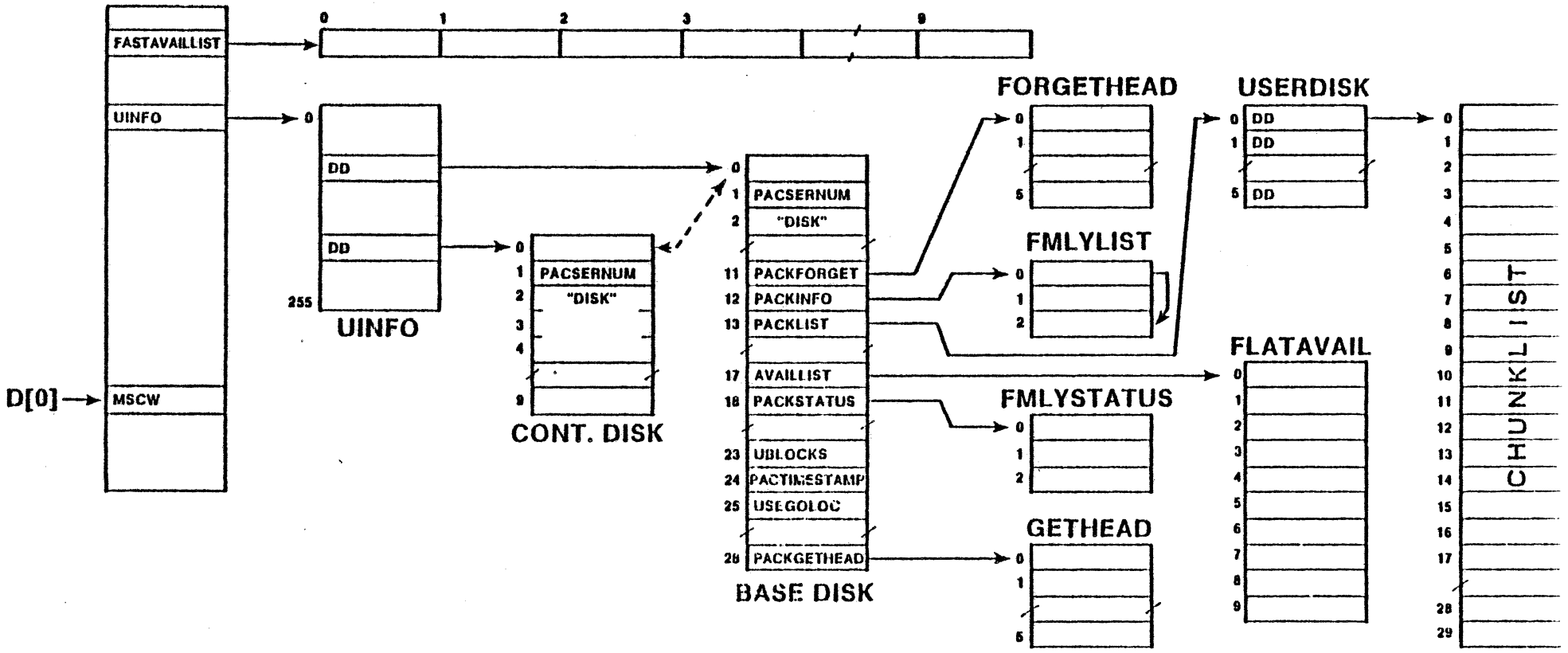

FASTAVAILLIST
UINFO
MSCW
D[0]
DD
DD
UINFO
PACSERNUM
"DISK"
CONT. DISK
PACSERNUM
"DISK"
PACKFORGET
PACKINFO
PACKLIST
AVAILLIST
PACKSTATUS
UBLOCKS
PACTIMESTAMP
USEGOLOC
PACKGETHEAD
BASE DISK
FORGETHEAD
FMLYLIST
FMLYSTATUS
GETHEAD
USERDISK
DD
DD
DD
FLATAVAIL
CHUNKLIST

Figure 8-11. Available Disk List

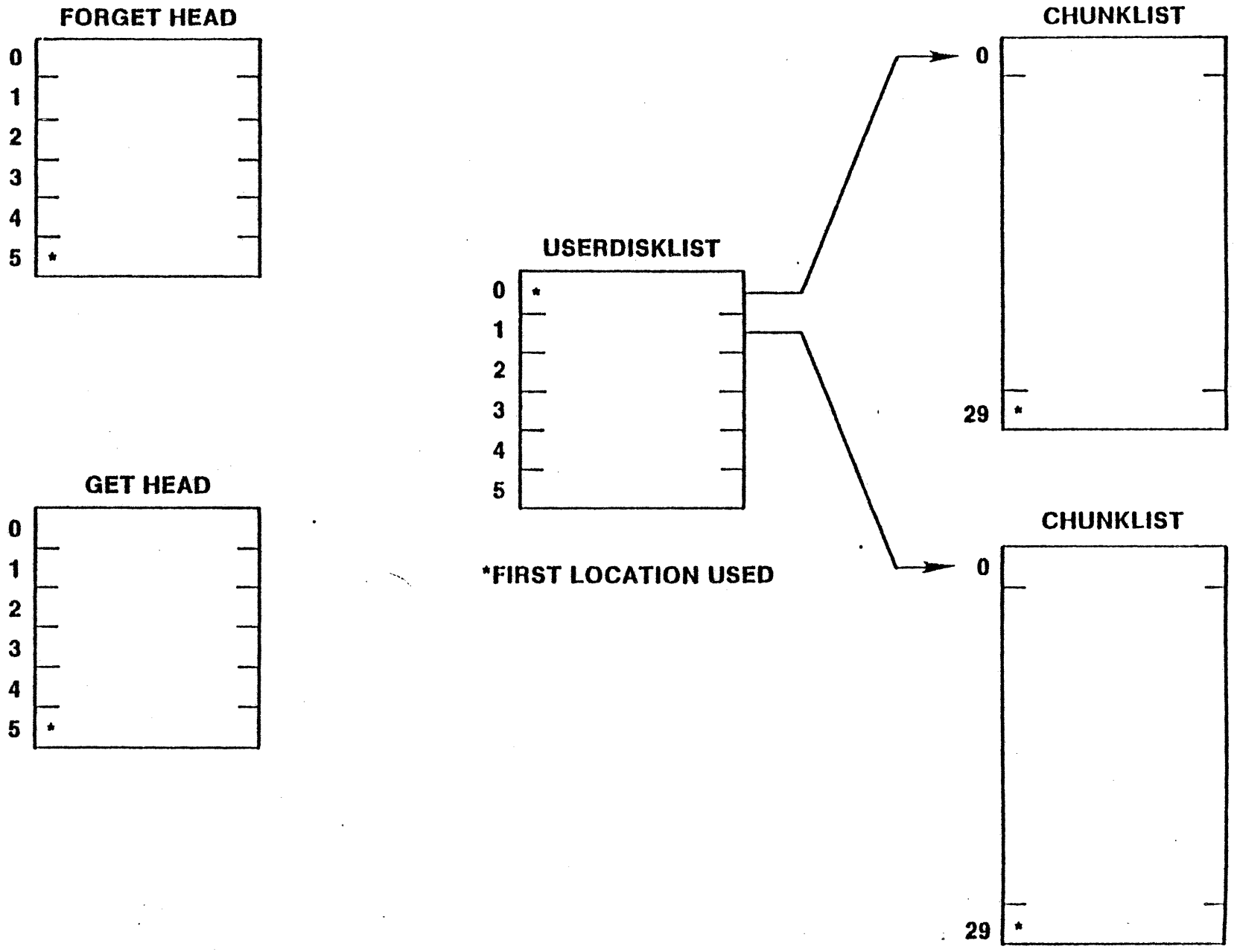


Figure 8-11. Available Disk List

INDEX	CONTENT
	UDI/FI/SIZE
0	0 000000 000000
?	
9	0 000000 000000
A	0 000000 4000A1
B	0 000100 4002BC
C	0 000200 40D6BC
D	0 000300 80007D
E	0 000400 81D802

This array may be interpreted as follows:

Family member 2 is associated with two CHUNKLISTS (chunklists 4 and 3). Out of the 30 possible variable size chunks in CHUNKLIST 4, the largest size chunk contains 4""1D802"" segments and out of the 30 possible chunks in CHUNKLIST 3, the largest chunk contains 4""7D"" segments.

Family member 1 is associated with three CHUNKLISTS, CHUNKLISTS 2, 1 and 0. CHUNKLIST 2's largest chunk contains 4""D6BC"" segments. CHUNKLIST 1 and 0 contain largest chunks of 4""2BC"" and 4""A1"" respectively.

Note:

> Notice that within a given family index, the chunk sizes vary in descending order.

Assume that GETUSERDISK is requested to get 4""100"" segments of disk from family member 1 (using the table above). GETUSERDISK will do a MASKSEARCH on GETHEAD looking for member 1's area. Once the area is found, a row address word is built and returned to the calling procedure. The GETHEAD array and associated CHUNKLIST will be updated. (The FORGETHEAD must be updated if the lowest address chunk was used).

The FORGETHEAD array is used to update appropriate CHUNKLISTS when disk areas are no longer required by the user program (or MCP).

DISKMAPPER

DISKMAPPER is called during system initialization to set up the available disk tables and complement the FLAT DIRECTORY. Actually, setting up of the available disk tables is a natural consequence of directory complementing. During a run of DISKMAPPER, two other tables are built:

FMLYLIST

FMLYSTATUS

These two arrays are shown in figure 8-10 and will be explained in the following paragraphs. Directory complementing and the available disk tables will be discussed last.

FMLYLIST

FMLYLIST, called PACKINFO on memory dumps, contains information about the family members (one word per member) and is indexed by family index. This array is first set up in READADISCLBL (before the call on DISKMAPPER) and simply contains the family list information starting in absolute segment 8 on the base family member. Word 0 will be the number of family members. It is DISKMAPPER's responsibility, regarding this array, to include additional information in each member's word. All words in FMLYLIST, other than word 0, will contain the following after DISKMAPPER is through with the array:

```
      PBITF = 47:1,    % 1 if family member present.
    DKBACKUPF = 46:1,  % 1 if contains BACKUP DIRECTORY.

  DKGOINGFLAT = 45:1,  % 1 if getting a FLAT DIRECTORY.

       IADPKF = 44:1,  % IAD pack.

     DKEUNBRF = 43:8,  % EU o of family member.

     DKSPEEDF = 31:4,  % Speed attribute (0 for pack).

DKTABLEINDEXF = 35:4,  % Index into DISKTABLE.

DKAREACLASSF  = 27:8,  % AREACLASS attribute.

DKSERIALNBRF  = 19:20  % Members serial number in binary.
```

The information that DISKMAPPER adds to FMLYLIST is in the fields:

DKSPEEDF

DKTABLEINDEXF

DKAREACLASSF

The DKAREACLASSF will be the class of disk, if classed. This info was transferred from the UNITADDL array (set up at cold start time). DKSPEEDF and DKTABLEINDEXF info came from the value array DISKTABLE. The layout of DISKTABLE can be found

in the MCP.

FMLYSTATUS

FMLYSTATUS, called ""PACKSTATUS"" on memory dumps, is also indexed by family index but the first word does not contain the number of family members as does its associated array, FMLYLIST.

FMLYSTATUS words are set up mostly by DISKMAPPER but don't really contain much information. Basically, this array contains info indicating which storage units/packs on a family member are either not ready or in a write lock-out condition. All words but word 0 have the following format:

```
DKCLASSCNTF = 41:22, % number of classed segments.

DKSWITCHESF = 19:20; % if on SU/PK is locked out.
```

DIRECTORY COMPLEMENTING AND THE ""AVAILABLE"" TABLES

DISKMAPPER is usually the procedure thought of when directory complementing comes to mind. Actually, DISKMAPPER calls the global procedure ""FLATREADER"" and it is this procedure that really does the complementing.

During the general process of complementing the directory, three things are done:

1. Verify that all flat headers are good.

2. Build the family's ""FLATAVAIL"" table.

3. Build the family's ""DISCAVAIL"" table.

The avail tables are built as a natural consequence of checking the FLAT DIRECTORY headers. Each time a header is read that has the AVAILMARK in it (4""3C3C"") an entry is made in the FLATAVAIL table. As far as the DISCAVAIL table is concerned, it is assumed that all possible disk is available to the family being checked. Each time a header is found in the FLAT DIRECTORY, all of its rows are removed from the available table. When directory complementing is finished, all that will remain in the available table is available user disk.

DISKMAPPER/FLATREADER

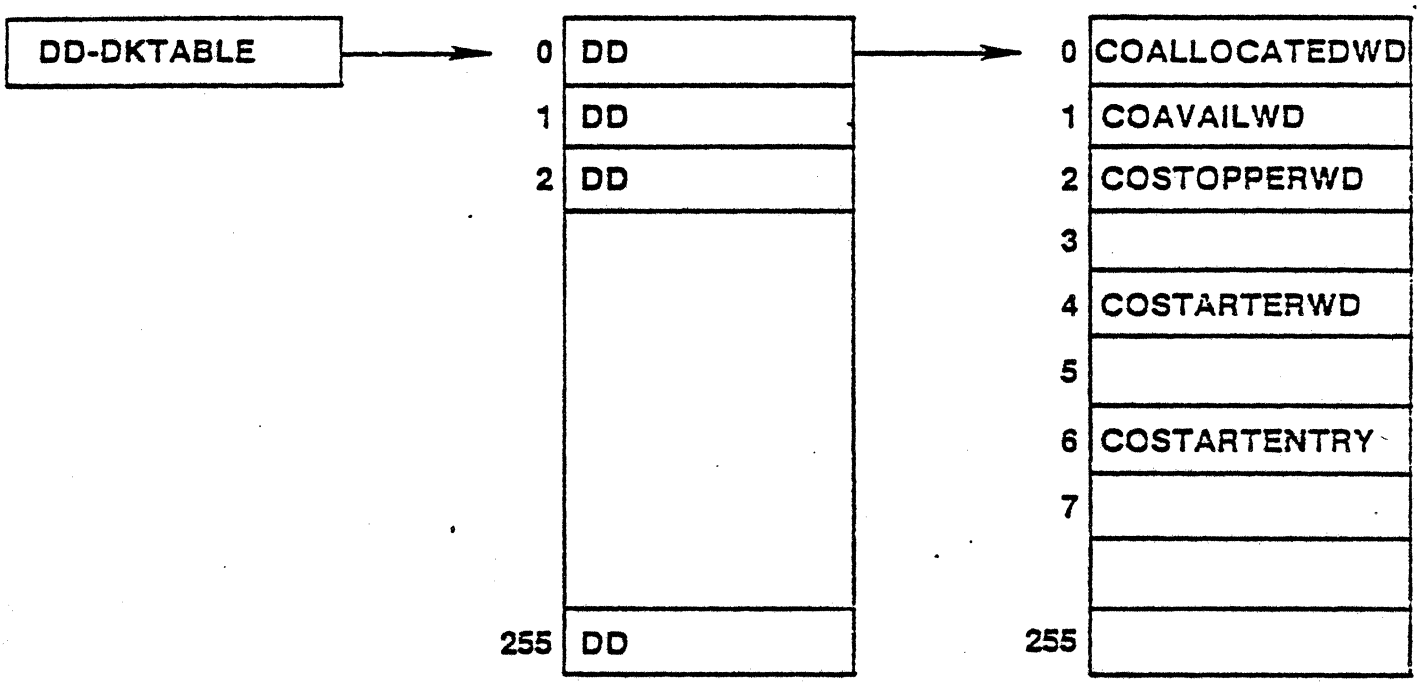
To accomplish the tasks above, DISKMAPPER builds an array named ""DKTABLE"" (known as ""LINKLIST"" in FLATREADER) and passes this array as well as a code to FLATREADER. The code tells FLATREADER to rebuild the FLATAVAIL table (REBUILDFLATAVAILF) and to rebuild the DISCAVAIL table (REBUILDDISCAVAILF). By implication, the headers will be checked for validity. When FLATREADER has done its part in this process, DKTABLE will contain the list of available areas and the FLATAVAIL will be completely finished and ready for use.

Return from FLATREADER, DISKMAPPER constructs the DISCAVAIL tables shown in figure 8-11 from information in the DKTABLE that was previously passed to FLATREADER.

Figure 8-12 shows the initial set up of DKTABLE as constructed by DISKMAPPER and passed to FLATREADER. FLATREADER calls the global procedure TAKEBACKDISC to make the entries in this table and it is TAKEBACKDISC that should be referred to if more information is desired on this table.

FLATAVAIL TABLE (STRUCTURE)

FLATAVAIL is the array containing available FLAT DIRECTORY header locations. This array is actually composed of two lists and a misc. part. To analyze the array, the following example is given:



WORD IN DKTABLE		INITIALIZED TO
0	COALLOCATEWD	000000 000006
1	COAVAILWD	
2	COSTOPPERWD	FFFFFF F00003
3		000010 000000
4	COSTARTERWD	000000 000006
5		
6	COSTARTENTRY	<TOTAL SEGS ON UNIT> 00002

Figure 8-12. Dktable

WORD IN DKTABLE		INITIALIZED TO
0	COALLOCATEWD	000000 000006
1	COAVAILWD	
2	COSTOPPERWD	FFFFFF F00003
3		000010 000000
4	COSTARTERWD	000000 000006
5		
6	COSTARTENTRY	<TOTAL SEGS ON UNIT> 00002

Figure 8-12. Dktable

INDEX	CONTENT
0	0 000000 000006
1	0 000000 00000B
2	0 FFFFFF F00002
3	0 FFFFFF F00003
4	0 000000 00000C
5	0 000000 000164
6	0 000009 30000A
7	0 000016 400003
8	0 000021 300002
9	0 000000 900007
A	0 00001F A000008
B	0 000000 600009
C	0 000000 000000

The format of the table given above has been redrawn in figure 8-13 to give a better idea of its parts and their association. The three parts are:

1. Address list

2. Size list

3. Miscellaneous

The address and size lists are ordered with the smallest number first. Each entry in the address list is an index in segments into the flat directory of where an available header location may be found. The entries in the size list give the number of segments in each area.

There are two locations in the miscellaneous area. LARGIX contains the size of the largest available contiguous area that may be obtained. Compare LARGIX in figure 8-13 to the size list entries of this figure.

The AVAIL word in the miscellaneous part points to the next available location in the list.

ADDRESS LIST PART

NAME	INDEX	CONTENTS
ASTARTER	0	6
FIRSTUSED	6	93 - A
	A	1FA - 8
	8	213 - 2
ASTOPPER	2	FFFFFFF - 2

SIZE LIST PART

NAME	INDEX	CONTENTS
SSTARTER	1	B
	B	6 - 9
	9	9 - 7
	7	164 - 3
SSTOPPER	3	FFFFFFF - 3

MISCELLANEA PART

NAME	INDEX	CONTENTS
LARGIX	5	164
AVAIL	4	C

Figure 8-13. Flat Header Available List

DIRECTORY COMPLEMENTING, AN OUTLINE OF THE PROCEDURES

STARTSYSTEM

Wait for CONTROLLER to start running so the operator can see spo messages.

Fork READADISCLBL for each label to be read. READADISCLBL reads in the disk label and moves nearly all information from the label to the unit's UINFO array. The label information starting in segment 8 is placed in FMLYLIST (see figure 8-10). This procedure links all family members together by use of UINFO word zero.

Wait till all READADISCLBL's are finished.

Call VERIFYFAMILY to read in the header for SYSTEMDIRECTORY. This is the header that describes the FLAT DIRECTORY for the family and is found by use of USEGOLOC (previously set up by READADISCLBL in the unit's UINFO array). This call on VERIFYFAMILY will also result on a call on DISKMAPPER. DISKMAPPER will set up all available disk tables (see figures 8-11 and 8-13) for the H/L family and, by use of FLATREADER, will complement the family's FLAT DIRECTORY.

Call VERIFYFAMILY again but this time VERIFYFAMILY will call FLATREADER to find the header for the ACCESS STRUCTURE. This header is found by sequentially reading through the FLAT DIRECTORY until the correct header title is found. If SYSTEM/ACCESS does not exist.

Call INITIALIZECATALOG to buildaheader for the ACCESS STRUCTURE and up the global array fastavaillist. INITIALIZECATALOG will also get the first row of the structure and initialize the PAST portion of it.

Call VERIFYFAMILY to verify the H/L unit's PAST. At this point it will be found that the family doesn't have an entry in the PAST (we just initialized it) and therefore, CREATEFAMILY is called to make the family entry in the PAST.

Call FLATREADER to rebuild the family's FAST. FLATREADER will read the entire FLAT DIRECTORY for the family. When an available header location is

found, it is entered in the availlist (see figure 8-10). When a file's header is found, FILEHANDLER is called to enter the file in the family's FAST block(s).

Call FILEHANDLER to enter the ACCESS STRUCTURE's header into the FAST.

Put the location of the access header in seg 0 of the FLAT DIRECTORY (DIRFRSTCATALOGLOCWORD

If SYSTEM/ACCESS does exist:

Call FLATREADER to initialize the global array FASTAVAILLIST.

Remove all off-line units from the PAST.

If a given family is not in the PAST then call CREATEFAMILY to place an entry in the PAST and make a note in the UINFO ARRAY (rebuilding) to let STARTSYSTEM know that the FAST should be rebuilt.

If the FAST should be rebuilt, call FLATREADER to do it.

Fork INITIALIZEINTRINSTUFF to set up the INTRINSICS STACK.

Let CONTROLLER know that it's okay to use the directory.

Restore working set factors and fork WSSHERRIFF if OLAYGOAL is greater than 0.

Start supervisor program.

Wait until JOBDESC is ready.

SECTION 9

PROCESS CONTROL

INTRODUCTION

The control and handling of programs to be run (and running) has many aspects. Initiating a program includes locating the code file, building the process stack, bringing code into memory, etc. Programs may cause other programs (tasks) to be executed and affect those programs once they are running. Other programs (jobs) are run to control the sequence of tasks. Backup files must be printed or punched when jobs come to an end. Also, how one program may influence another (events) must be considered. These topics and others are the concern of this section.

Due to the wide area that process control covers, this section has been divided into 6 sub-sections:

1. PROGRAM INITIATION AND TERMINATION

2. JOBS

3. CONTROLLER

4. AUTOBACKUP

5. MISCELLANEOUS PROCESS CONTROL PROCEDURES

6. PROCESSCONTROL, A PROGRAM

The first sub-section, PROGRAM INITIATION AND TERMINATION, discusses how process stacks are set up, code located, etc. The second sub-section is about JOBS and associated MCP procedures such as CONTROLCARD and WFL. A job is defined in this text as any program compiled by the WFL compiler. The third sub-section explains CONTROLLER's involvement in JOB QUEUES and this is followed by a discussion on AUTOBACKUP and how it relates to JOBS, CONTROLLER and JOBFORMATTER. The next sub-section, MISCELLANEOUS PROCESS CONTROL PROCEDURES, discusses events and also provides information about specific MCP procedures such as GEORGE, WAITP, CAUSEP and TIMETUNNEL. The last sub-section is not that appropriate in this manual

but it does discuss an example ALGOL program using events and intrinsics. The program will occasionally be referenced but it has been assumed the reader is already aware of the use of events and intrinsics and this sub-section could possibly be ignored by the knowledgeable reader.

PROGRAM INITIATION AND TERMINATION

Program initiation consists of building a Program Information Block (PIB) and process stack for the program to be run. The code file must be located, the segment dictionary set up and a log entry made. Somewhere in this sequence of events, parameters must be passed to the program (if it gets any), a mix number assigned and linkage established between the new task and its parent task and siblings if any exist. Once a program is generally set up, the stack number is placed in the READY QUEUE. It is at this point that an available processor will move to the new stack and exit down into the program.

When the final exit of a program is executed, the MCP takes over again and makes another log entry, continues any jobs or tasks waiting on the completion of the present task and makes all resources available to the system. Since the MCP was doing all this work on the terminating stack, it would not be proper to do a FORGETSPACE on the stack at this time. Because of this, a TERMINATE entry is placed in the queue for ETERNALIR and the event for ETERNALIR caused (to waken ETERNALIR). It is ETERNALIR (because of the TERMINATE entry) that does a FORGETSPACE on the process stack itself.

INITIATING PROCEDURES

The first array to be obtained for any program is its PIB. This array's memory space is obtained in ANABOLISM for independent runners, in JOBSTARTER for jobs and in MUTATE or INITIATEUSERTASK for tasks. The variables within the array are set up by various MCP procedures and some locations within the array may be accessed and changed by the associated program(s) once in execution. After a PIB has been partially set up, the MCP procedure, DOCTOR, is called to locate the code file on disk and read in the file's header. Once the header is in memory and a descriptor pointing to it is placed in the DISK FILE HEADER STACK, DOCTOR places its COREINDEX in the TASKINFO word in the PIB. COREINDEX is the MCP name for the index of a descriptor into the DISK FILE HEADER STACK. Having set up its part of the PIB, DOCTOR calls INITIATE.

It is INITIATE's responsibility to get space for a process stack and set up several locations in the process stack. The process stack and PIB as set up by DOCTOR and INITIATE can be seen in figure 9-1. The process stack is made to look to the hardware as though it had always been running. Seeing how the

stack is initially constructed, one would assume that the stack would be moved to and then an EXIT instruction executed. This is, indeed, the case but notice, of equal importance, that no mention has been made of reading the program's segment dictionary into memory and getting it set up. In fact, this has not been done and what is shown in figure 9-1 is the program as it is when its stack number is placed in the READY QUEUE.

When INITIATE sets up the process stack, a stack number and mix number are obtained by use of the procedure, PICKASTACK. The mix number is placed in the SERIAL word in the PIB and a descriptor pointing to the stack is placed in the STACK VECTOR ARRAY at the appropriate location. The location in the stack of figure 9-1 referred to as PFL is actually named PROCESSFAMILYLINK and it is this location that is used to link associated tasks together.

The MCP procedure, GEORGE, is used to look at the READY QUEUE. If "GEORGE" sees that the stack in the READY QUEUE should be moved to, the procedure, STACKMOVER, is called. STACKMOVER has only 2 instructions:

```
MVST

EXIT
```

The MVST instruction will cause the hardware to build a TOSCW from current register settings and store in the first word of the stack it is presently on. Next, the descriptor for the new stack (from the STACK VECTOR ARRAY) is fetched and BOSR and LOSR are changed to point to the new stack. Finally, the hardware RDLK's the TOSCW of the new stack, replacing it with the BOX NUMBER of the executing processor and using the fetched TOSCW to set up the S and F register and, in the example of figure 9-1, D[1]. Figure 9-2 shows the register/stack relationship after the MVST is complete.

When the EXIT instruction is executed, stack cutback and display update occur. At this time, the instruction pointer is also changed and PREBOJ starts executing. PREBOJ is an entrypoint into GEORGE and is used to initialize a few time variables for the program. When PREBOJ exits, it will be into NORMALBOJ and it is this procedure that locates the program's segment dictionary, reads it into memory and sets up appropriate MSCW linkage.

Figure 9-3 shows memory for the program after NORMALBOJ has finished its job but before the exit into the program. Notice how the MSCW immediately above the DUMMY RCW has been changed and is now pointing to the MSCW in the segment dictionary. The top RCW in the stack was made by NORMALBOJ by changing the TAG of the PCW in the segment dictionary to a 3 and then writing it into the process stack at the location shown.

The MSCW's shown in figures 9-1 through 9-3, that point at the PIB MSCW, do so by use of "PSEUDO-STACKS". Suffice it to say at this time that linkage can be established that will cause the hardware to point its D[1] register at the MSCW in the task array even though the task array is not a stack. PSEUDO-STACKS are also used with FILE INFORMATION BLOCKS (FIB's) and will be discussed in detail in section 10 in relation to this subject.

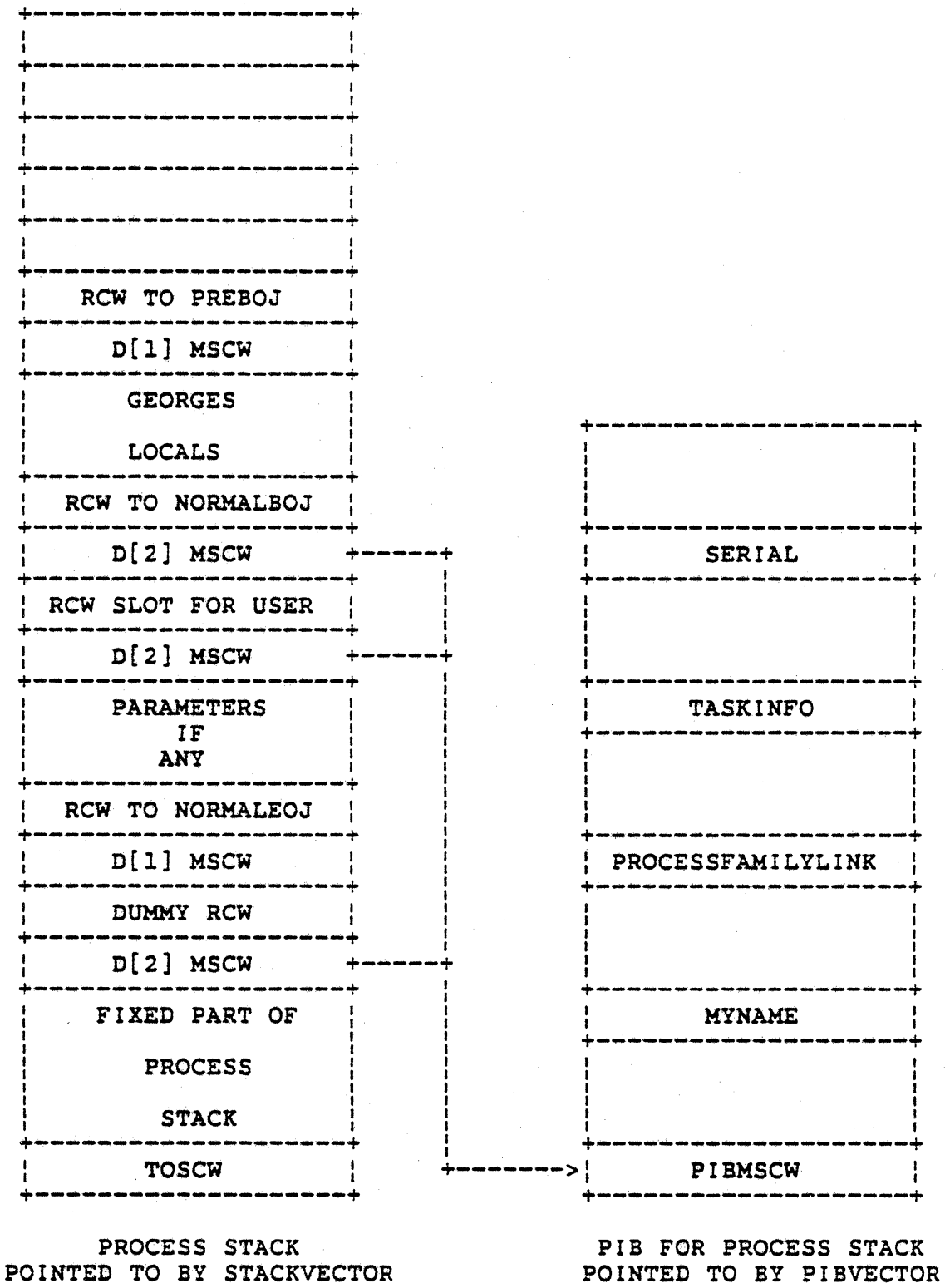


Figure 9-1. Process Stack Before First MVST Operator

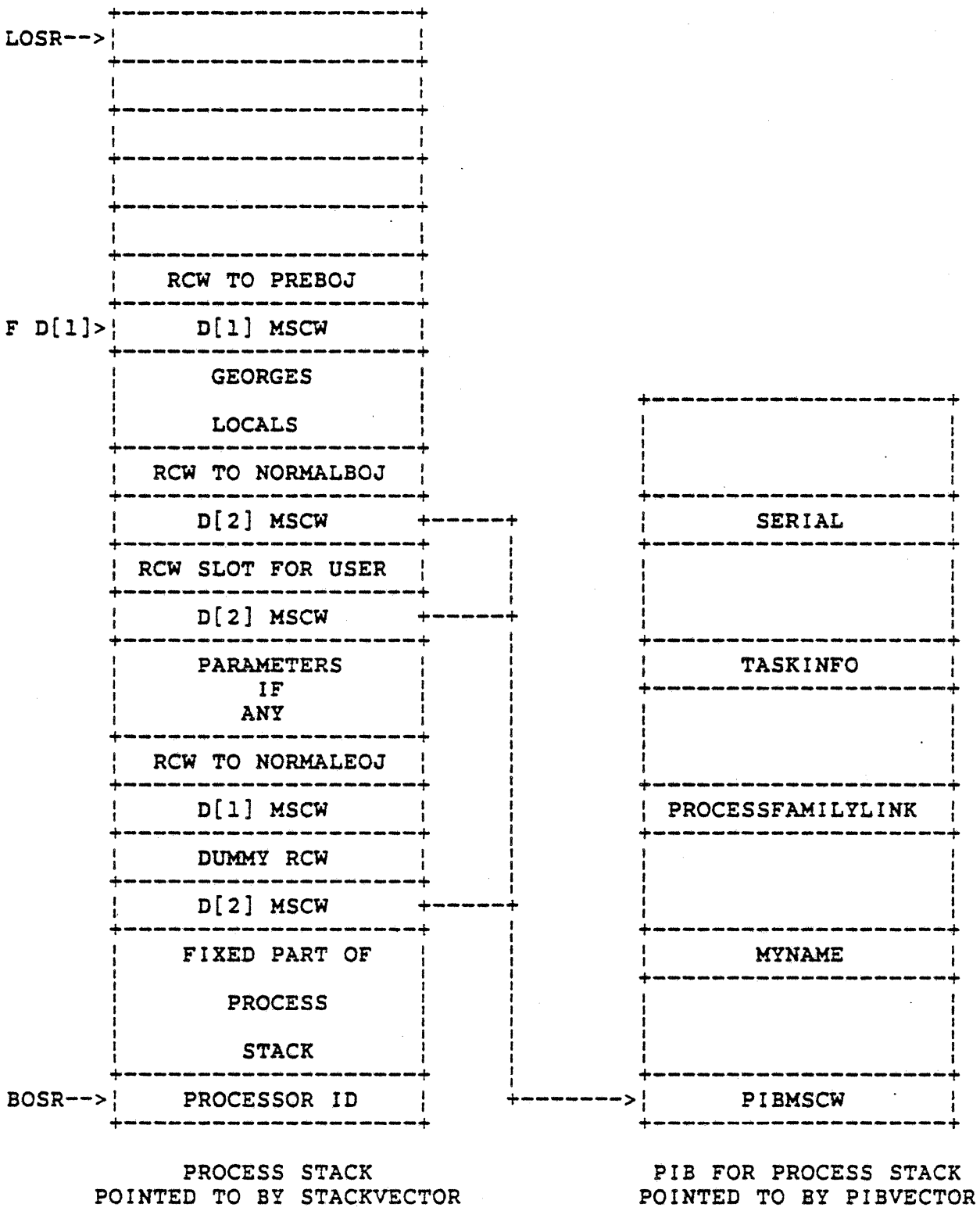


Figure 9-2. Process Stack After MVST Operator

TERMINATING PROCEDURES

When the final exit is made from a program into NORMALEOJ, stack cutback and display update occur as shown in figure 9-4. The segment dictionary now appears to be dissociated from the process stack but this is not the case since its stack number is saved in a field in a word in the PIB. This word, named CODELINKS, contains the process stack's segment dictionary number and the associated intrinsic stack number. When NORMALEOJ starts executing, a log entry is made, the segment dictionary space forgotten (along with other arrays in the fixed base portion of the process stack) and FORGETUSERDISK is called on the OVERLAY header (OLAYFILEDESC). Note that the segment dictionary space is not forgotten if other processes are still using it. The number of processes using a segment dictionary stack is maintained in the PIB of the segment dictionary stack.

The last 2 arrays in use before NORMALEOJ finishes are the PIB and the process stack. Since D[1] is currently pointing at the PIB, it is better to change it before FORGETSPACE is called on that array. In reference to figure 9-5, the following occurs:

1. Tell ETERNALIR to terminate the stack.

2. Call GEORGE and tell him (via the MANDATE parameter) to get out of the stack.

ETERNALIR will get the terminate message. ETERNALIR will call TERMINATE. TERMINATE will call FORGETSPACE on the process stack. In addition, it will release the memory area for the PIB if it was system generated.

JOBS

Jobs are those programs compiled by the WORK FLOW LANGUAGE (WFL) compiler and may enter the system via a card reader or ODT. After a job is compiled, it is entered into a JOB QUEUE by CONTROLLER. This aspect of jobs will be covered in the following sub-section.

The top left block shown in figure 1-6 (in section 1 of this manual) represents a job deck in a card reader. When the reader is made ready ETERNALIR will see a reader has changed status (gone from not ready to ready). Seeing that a card reader has gone ready, ETERNALIR will fork CONTROLCARD and then go back to sleep on ETERNALIREVENT. CONTROLCARD is the MCP procedure that provides the interface to the WFL compiler and it is the procedure that calls WFL.

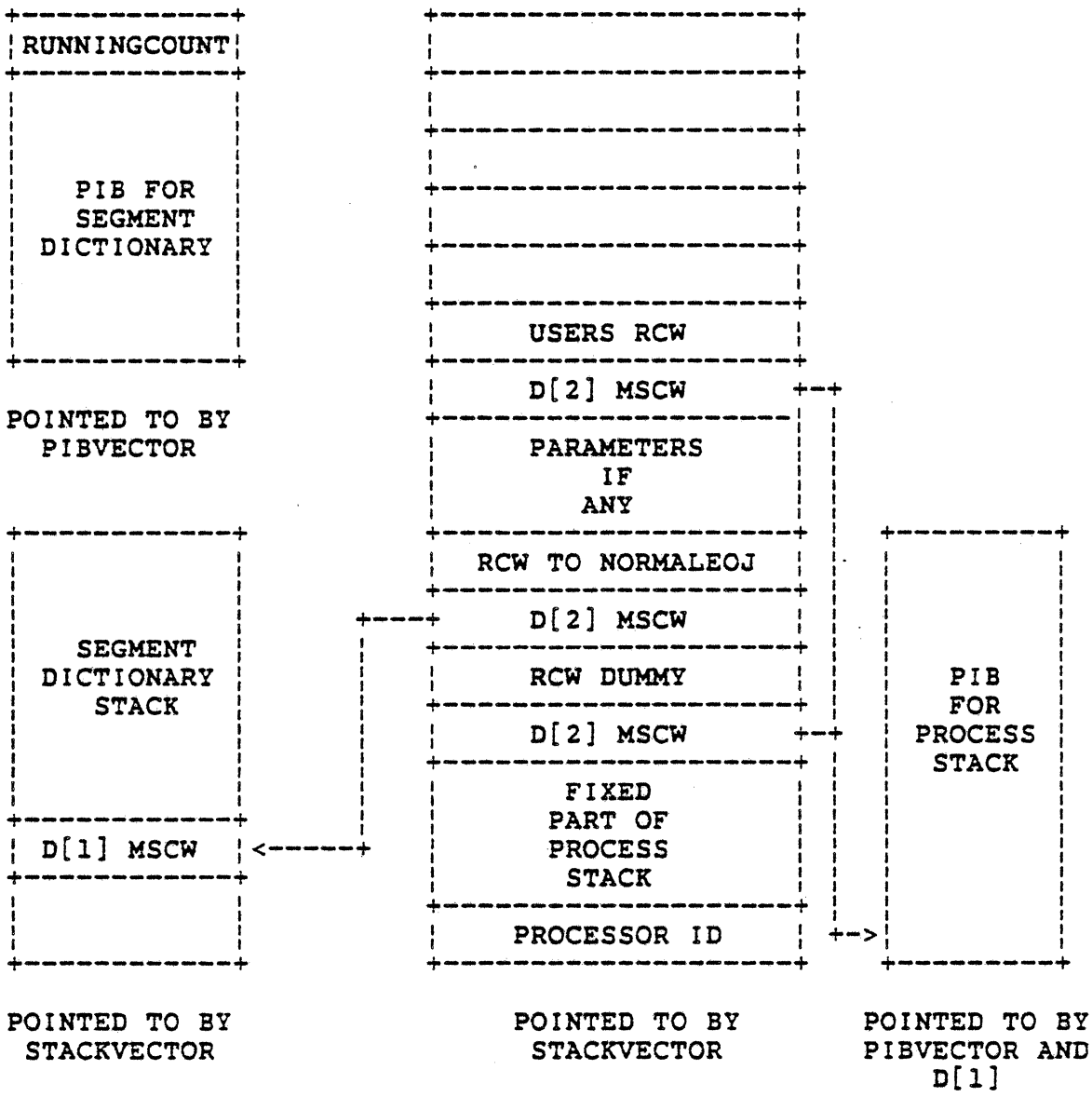


Figure 9-3. BOJ Responsibilities

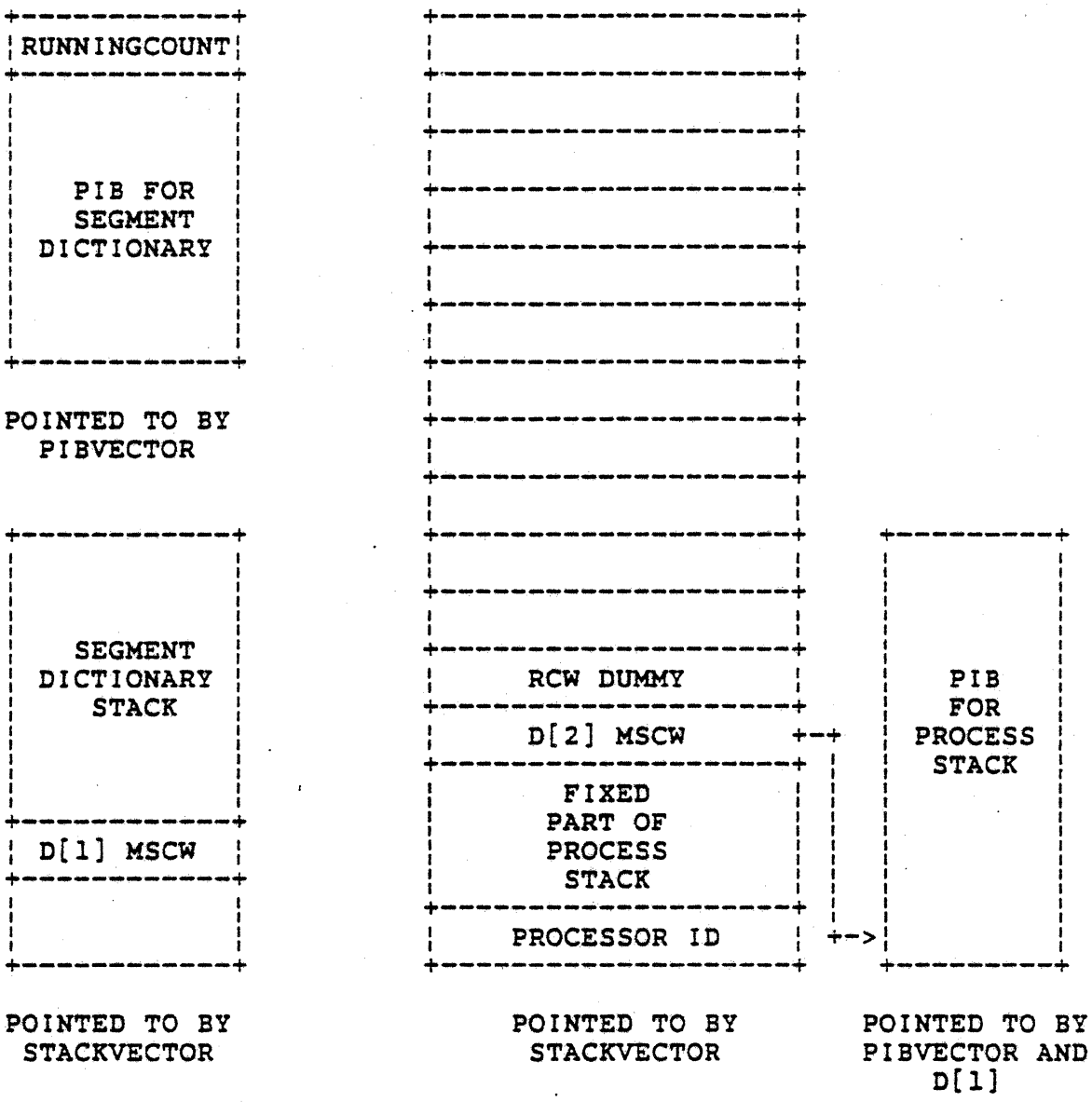


Figure 9-4. NORMALEOJ Responsibilities

WFL COMPILER

The WORK FLOW LANGUAGE compiler is a DCALGOL program that is bound into the MCP. When the program is called by CONTROLCARD it starts reading cards from the designated reader and compiles a code file quite similar to that of ALGOL or COBOL, etc.

The WFL compiler will not go to end of job until all jobs placed in the card reader have been compiled. As each code file is completed, the disk file is locked. The MCP procedure CLOSE (responsible for locking files) takes special action when closing JOBFILES. When a JOBFILE is closed, this procedure places a SCHEDULEREQUEST entry in the CONTROLLERQ and leaves the header in the DISK FILE HEADER STACK. Responding to the SCHEDULEREQUEST, CONTROLLER will place an entry for the new job in a JOB QUEUE. When the last job has been compiled, CONTROLCARD/WFL leaves the mix.

Figure 9-6 shows the layout of the WFL compiler symbolic listing. There are only 2 first level procedures within the compiler and all the outer block does is selects one or the other. If WFL was called with a FILECARDV (card reader for instance) then BUNCHAJOBS is called. In all other cases, MUTATECASE is called.

BUNCHAJOBS is the procedure responsible for looping on all jobs in the card reader. BUNCHAJOBS reads the first card and if the card is:

```
? JOB BEGIN
```

then it's an old style (pre I.9) syntax and OLDJOB is called to compile it. If the card reads:

```
? BEGIN JOB
```

it's the new syntax and JOB is called instead.

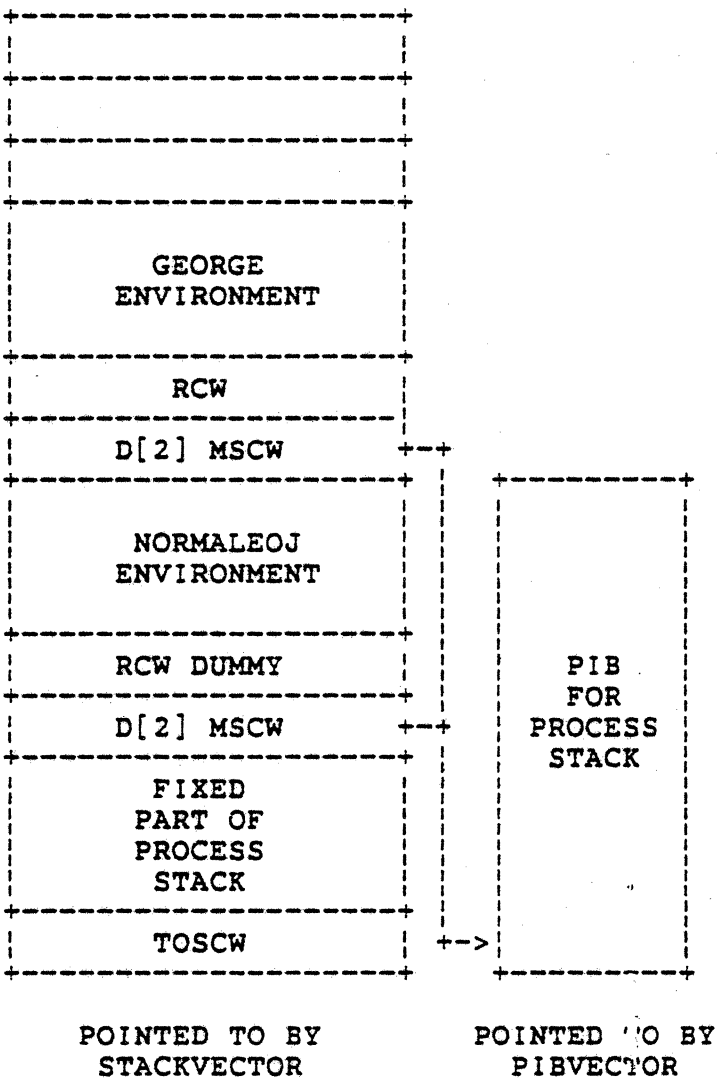
Also a part of the WFL symbolic are 3 separate procedures:

1. CCSTRINGCONV

2. CCSTRINGFUNCTION

3. CCVARIABLEPPB

The first procedure is for HEX, OCTAL and DECIMAL functions, the second is for the STRING function and the last is for mapping string variables into PROGRAM PARAMETER BLOCKS (PPB's) and FILE PARAMETER BLOCKS (FPB's).



A termination request has been linked into the queue for the ETERNALIR running in the BOX this stack is in. ETERNALIR will call FORGETSPACE for the STACK and PIB. GEORGE was called to move the processor to another STACK.

Figure 9-5. STACK Termination

```
WFL/MCP interface globals

WFL
  MUTATATECASE
        - handles task-FILECARDS and ?FA
  BUNCHAJOBS
        I/O and error routines
        SOMEKINDAJOB
           LABELACARDREADER
           code and pool-data emitters
           FINISHAJOBPPB
           OLDJOB - handles "old" syntax
              CCTABLES
              utilities
              statements
              block control
            JOB - handles "new" syntax
              CCTABLES
              utilities
              statements
              block control
        outer block of SOMEKINDAJOB
              - select type of job based on first token
        outer block of BUNCHAJOBS
       - loop for all jobs on the input device
  outer block of WFL
       - select either MUTATECASE or BUNCHAJOBS
```

Figure 9-6. Structure of WFL Symbolic

JOBFILES

Figure 9-7 is that of a typical JOBFILE. What is normally found in code files is the fixed format SEGO, the PPB, segment dictionary and object code. In addition to this, JOBFILES contain the source language card images, spooled card files, logging space and a place for the process stack to be saved.

Within SEGO of a job's code file, the following words are important:

WORD 3 Points to the logging space.

WORD 12 Points to the job's PPB.

WORD 18 Points to the segment dictionary.

WORD 27 Points to the process stack.

The word numbers given above are in decimal and consider 0 significant.

Card images start in segment 1 and are linked together by the first word in each segment. The second word in each segment is the number of characters in the card image. The image, itself, starts in the third word.

The TASK's PPB and DECK INFO segments are data pools and are pointed to by descriptors generated from the job's stack building code.

CONTROLLER

CONTROLLER, like WFL, is a DCALGOL program bound into the MCP. This program, however, is an independent runner that is always in the mix. CONTROLLER has 3 major responsibilities:

1. Absolute control of JOB QUEUES.

2. ODT screen displays (ADM).

3. Executing certain operator messages (via SETSTATUS and GETSTATUS) such as PG, PD, etc.

This sub-section is mainly concerned with CONTROLLER's JOB QUEUE responsibilities.

Again, referring to figure 1-6, closing a JOBFILE will cause an entry to be placed n the CONTROLLERQ and thereby, sent to CONTROLLER. This entry contains a SCHEDULEREQUEST code and the COREINDEX of the code file's header in the DISK FILE HEADER STACK.

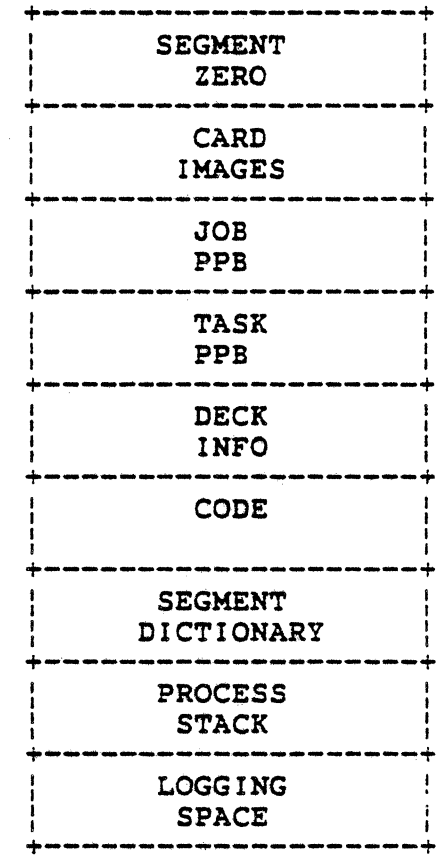


Figure 9-7. Typical JOBFILE

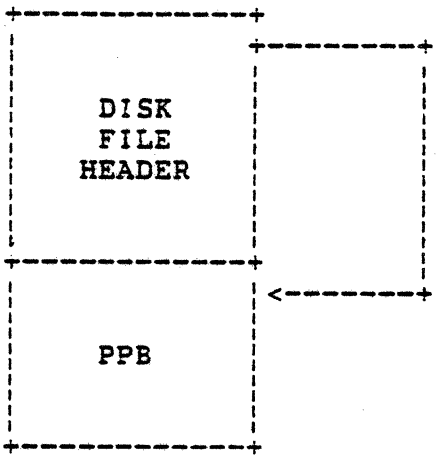


Figure 9-8. JOBDESC-JOB QUEUE ENTRY

CONTROLLER's outer block is essentially, a big case statement, After the message has been removed from the appropriate queue, CONTROLLER cases the code field in the message. A SCHEDULEREQUEST code causes CONTROLLER to call its local procedure NEWENTRY to create a JOB QUEUE ENTRY and place the entry in the correct queue.

A JOB QUEUE ENTRY is 1 array containing the JOBFILE's DISK FILE HEADER and PPB and is shown in figure 9-8. The first word in the array contains a field that points to the PPB and word 8, known as DISKLINK, is the word used to link this entry into the selected queue. JOB QUEUE ENTRIES are physically located in CONTROLLER's JOBDESC file. The block that is labelled "JOBDESC JOB QUEUES" in figure 1-6 has a dashed line leaving it that points to "JOBFILE CODE AND LOG INFORMATION." This line is actually coming from the headers in the JOB QUEUES of the JOBDESC file.

The NEWENTRY procedure within CONTROLLER starts execution by calling the MCP procedure GETJOBDESCRIPTION (passing COREINDEX). GETJOBDESCRIPTION returns the HDR/PPB combination which is subsequently written to the JOBDESC file at an available location. Next, NEWENTRY calls the CONTROLLER procedure ABSTRACT which sets up a special array in window format. (This array contains the job's attributes such as PROCESSTIME, IOTIME, etc.). ABSTRACT returns a QUEUE INDEX but note that "QUEUE INDEX" is not the same as "QUEUE NUMBER." NEWENTRY now calls another CONTROLLER procedure named QUEUEINSERT and passes the WINDOW array and QUEUE INDEX. QUEUEINSERT will insert the WINDOW array in the real WINDOW if the new job is of higher priority and then physically link the JOB QUEUE ENTRY (HDR/PPB) into a JOB QUEUE (this is where the DISKLINK word is used). The JOB ENQUEUING ALGORITHM is shown in figure 1-7 but will not be reviewed here.

After a job has been linked into a queue, the CONTROLLER procedure SELECTION, is called. This procedure looks at all queues and if a queue contains entries, the MIXLIMIT and TURNAROUND time are checked to determine if the job can be run. Assuming the job can be run, it is delinked from the queue and JOBSTARTER, an MCP procedure is called and passed the HDR/PPB of the job. JOBSTARTER builds a PIB for the job and calls DOCTOR passing this PIB to DOCTOR. DOCTOR calls INITIATE passing the PIB and INITIATE builds a process stack and places its number in the READY QUEUE.

Figure 9-9 shows a flow of the JOB DEQUEUING ALGORITHM.

ODT CONTROL CARDS

As mentioned earlier, JOBS can be entered into the system by use of card readers or ODT's. To work with an example,

consider the following statement entered on a ODT:

```
RUN X <etx>
```

When the message is entered and the ODT placed in transmit, the hardware receives a status change and calls the MCP procedure HARDWAREINTERRUPT. This procedure, seeing that it is a status change, forks KEYIN and KEYIN reads the ODT.

If the message is a primitive, then KEYIN executes the message but if it is not a primitive, the message is sent to CONTROLLER through the OPERATORINPUT queue for further interpretation. See figure 1-6.

CONTROLLER also has responsibility for parsing input text. When CONTROLLER sees the word RUN in an input message, the message is simply passed as a parameter to FORKCONTROLCARD and this procedure, as you may expect, forks CONTROLCARD. From this point on, job file control continues as described for card readers.

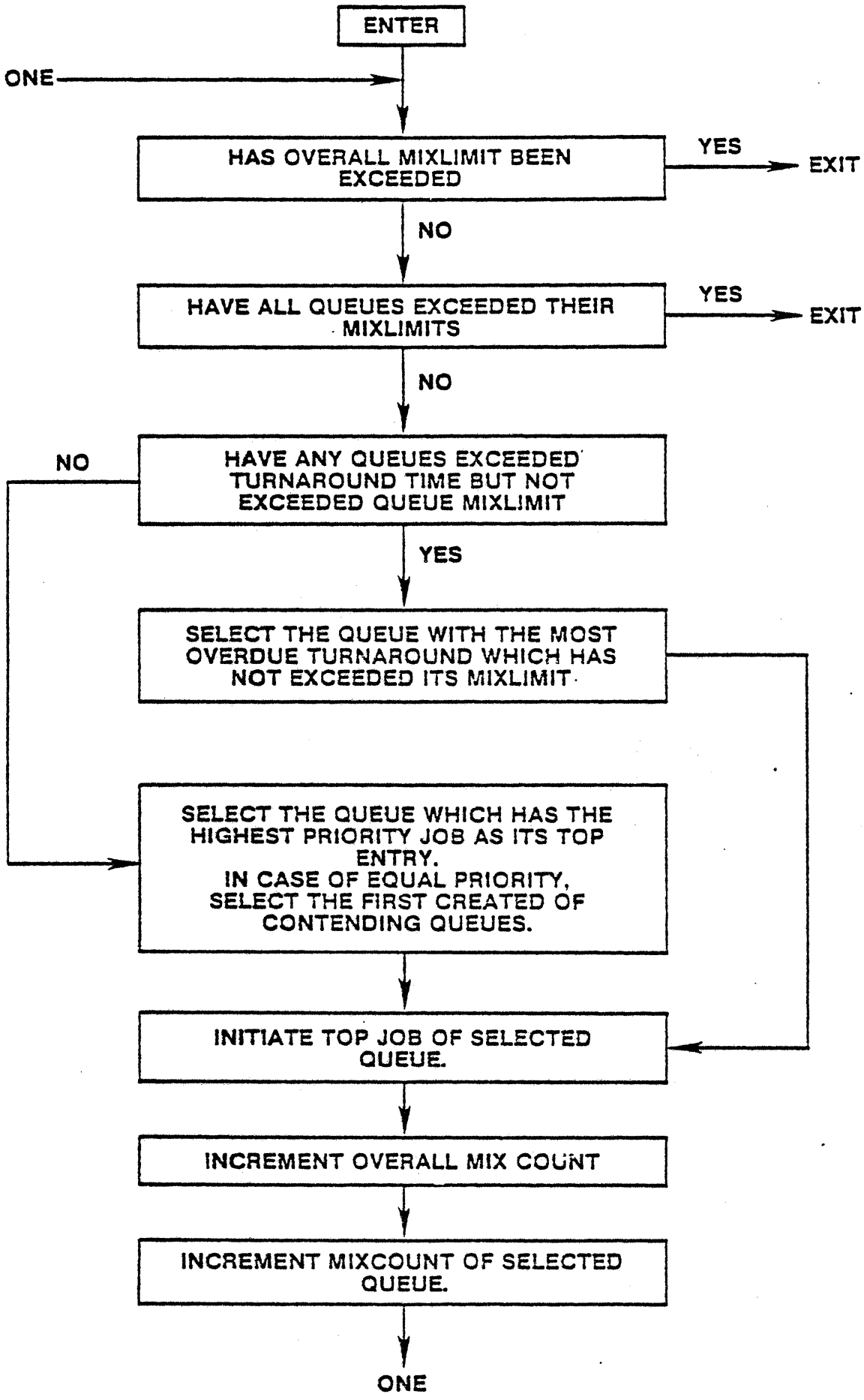

ENTER
ONE
HAS OVERALL MIXLIMIT BEEN EXCEEDED
YES
EXIT
NO
HAVE ALL QUEUES EXCEEDED THEIR MIXLIMITS
YES
EXIT
NO
HAVE ANY QUEUES EXCEEDED TURNAROUND TIME BUT NOT EXCEEDED QUEUE MIXLIMIT
NO
YES
SELECT THE QUEUE WITH THE MOST OVERDUE TURNAROUND WHICH HAS NOT EXCEEDED ITS MIXLIMIT
SELECT THE QUEUE WHICH HAS THE HIGHEST PRIORITY JOB AS ITS TOP ENTRY.
IN CASE OF EQUAL PRIORITY, SELECT THE FIRST CREATED OF CONTENDING QUEUES.
INITIATE TOP JOB OF SELECTED QUEUE.
INCREMENT OVERALL MIX COUNT
INCREMENT MIXCOUNT OF SELECTED QUEUE.
ONE

Figure 9-9. JOB DEQUEUING ALGORITHM

JOBDESC FILE

The JOBDESC file could be considered the FLAT DIRECTORY of JOBFILES since this is the only place these file's headers may be found on disk, but in addition to this, the JOBDESC file is used to save CONTROLLER OPERATION information, PERIPHERAL ASSOCIATION information, etc. These JOBDESC areas and others can be seen in figure 9-10.

SEGMENT 0 of the JOBDESC file is of fixed format, the layout of which is shown in figure 9-11. Words 4, 8, 14, 18 and 32 in SEGO are shown in circles in figure 9-10. The value found in these words is actually a word index into the JOBDESC file and not a segment index as might be expected. The numbers 0 through 449 in figure 9-10 are segment numbers.

The following information is more detail on the areas of the JOBDESC file and should only be glanced over. It was placed here to be near related information.

JOB QUEUE HEAD OF JOBDESC

Word	Comment
0	COPYFF = 39:20, number of valid words in this segment.
1	QUEUEFACTS array for queue 0. NAMEQF = 39:20, queue number (will be 0 for this word).
2-N	Words 2 through N have the same format as word 1. Each of these remaining words is associated with a queue. Word 2 is associated with queue 1, word 3 with queue 2, etc.

QUEUEFACTS ARRAY SAVED IN JOBDESC

The QUEUEFACTS arrays (Figure 9-14) are saved in the JOBDESC file to preserve them across halt/loads. The arrays for the various queues are found by use of the JOB QUEUE HEAD segment previously discussed. The layout of this array is given below:

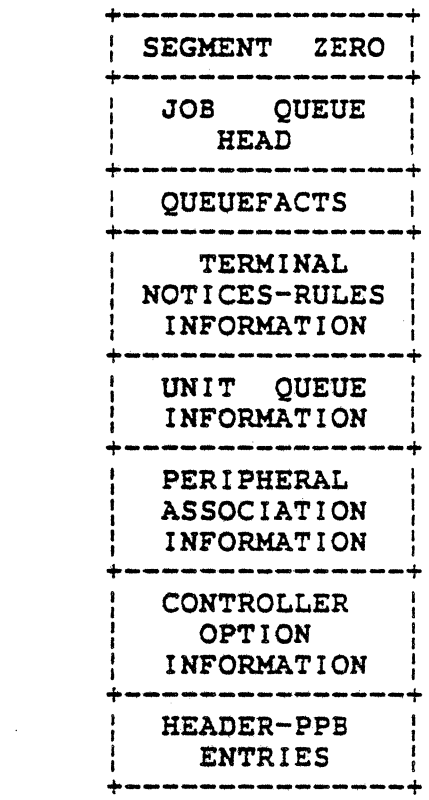


Figure 9-10. TYPICAL JOBDESC

```
SEG 0 IS INITIALIZE AT SEQUENCE NUMBER 80588000 IN CONTROLLER
WITH THE "BUFFER" ARRAY AS SET UP BELOW:

REPLACE POINTER (BUFFER) BY
   %        VALUE                                  WORD NUMBER    COMMENT
   %
      30,                  % SOLENGTH     %          0          30
      DEFAULTQ,            % SODEFAULTQ    %          1          0(VARIABLE)
      MCPLEVEL,            % SOVERSION     %          2          SEE BELOW
      0,                   % SORESERVED    %          3          0(RESERVED)
      JOBQADD * 30,        % SOJOBHEAD     %          4          1(VARIABLE)
      JOBQFOMATLENZ                        %          5          3
      JOBQHDLNG,                           %          6          SEE BELOW
      JOBQFOMAT1,                          %          7          1
      J * 30,                              %          8          SEE BELOW
      ADMFOMATLENZ,                        %          9          5
      ADMLENZ,                             %         10          150
      ADMFOMAT1,                           %         11          0
      ADMFOMAT2,                           %         12          0
      ADMFOMAT3,                           %         13          0
      K * 30,                              %         14          SEE BELOW
      UNITQFOMATLENZ,                      %         15          3
      UNITQLENZ,                           %         16          270
      UNITQFOMAT1,                         %         17          0
      N * 30,                              %         18          SEE BELOW
      PAFOMATLENZ,                         %         19          3
      PALENZ,                              %         20          300
      PAFOMAT1,                            %         21          0
      I * 30,                              %         22          SEE BELOW
      OPTIONFOMATLENZ,                     %         23          3
      OPTIONLENZ,                          %         24          60
      OPTIONFOMAT1,                        %         25          0
      TERMINATING                          %         26      48"FFFFFFFFFFFF"
      0,                                   %         27          0
      0,                                   %         28          0
      0;                                   %         29          0

      J = STARTING LOCATION (IN SEGS) OF THE TERM, ETC.
          INFORMATION IN THE JOBDESC FILE.

      K = STARTING LOCATION (IN SEGS) OF THE UNIT QUEUE INFO.

      N = STARTING LOCATION (IN SEGS) OF THE PERIPHERAL
          ASSOCIATION INFORMATION.

      I = STARTING LOCATION (IN SEGS) OF THE OPTION INFORMATION.

      JOBQHDLNG = JOBQHDLENZ              % 30
                  & MAXNUMQUEUES [15:8]   % 10

      MCPLEVEL = 0 & 7 [31:16] & 2 [47:16]
```

Figure 9-11. Segment 0 of JOBDESC File

Word 0 is set up as follows:

```
DISKJOBQLNG                    % 30
&(I + 1)                       % I = queue number minus 1

JOBQHDINDEXF                   % 39:20

&JOBQVALIDITY                  % 4

VALIDITYF                      % 47:8
```

All other words in this array have the same format as that of the core QUEUEFACTS array but word 0 of the core array is in word 1 of the disk array and word 1 of the core array is in word 2 of the disk array, etc.

TERM-NOTICES-RULES

The following information is kept here:

TERM controls number of lines to be displayed, width of lines, where first line should start, etc.

NOTICES -?

- 47:1 EVENTNOTICE
- 46:3 ADMSTATE
  - 0 ADMINACTIVE
  - 1 ADMACTIVE
  - 2 ADMSTOPPED
  - 3 ADMDELAYED (operator driven)
  - 4 ADMPOSTPONED (in use by a file)
- 43:1 TIMERUNNING
- 42:19 NOTICETIME
- 23:8 RULESLINK
- 15:8 LASTNOTICE
- 07:8 NEXTNOTICE

RULES. This is where information from the ""ADM"" message is saved.

```
  47:1 RULEINUSEF

  46:1 CONTINUEF

  45:1 LASTRULEFLAGF

  44:1 SKIPITF

  43:1 TOENDOFSCREENF

  42:1 DISPLAYRSVPMSGF

  41:1 DISPLAYDISPLAYMSGF

  40:1 SUPPRESSF

  39:16 DELAYF

  23:8 NEXTRULEF

 15:8 LINELIMITF

 07:8 RULETYPEF
```

UNIT QUEUE
---- -----

This area in JOBDESC is used to save information form the "UQ" ODT message. This message is used to associate units with queues.

Example 1:

```
     INPUT: UQ <etx>

     RESPONSE: QUEUE FOR CR10 IS 7
```

Example 2:

```
     INPUT: UQ CR15 6 <etx>

     RESPONSE: QUEUE FOR CR15 IS 6
```

Word 0 of the unit queue area:

```
     UNITQHEAD = <number of words in this area>

           & UNITQVALIDITYF % 8

           VALIDITYF & 47:8
```

Words 1 through N:

```
     UTYPEF = 46:7
```

```
UNITQNUMBERF = 23:34
```

For further information see "UQ" message in the WFL manual.

PERIPHERAL ASSOCIATION

This area in JOBDESC is used as a place to save "PA" ODT message information. The PA message is used to associate input and output devices. Only the following devices may be associated in this manner:

ODT

PRINTER

CARD READER

CARD PUNCH

TYPICAL INPUT:

```
PA CR10 = LP13 <etx>
```

RESULT:

Jobs started by CR10 will only print using LP13.

Word 0 in this area:

```
PAHEADER = (number of words in this area)
      & PAVALIDITY % 10
      VALIDITYF % 47:8
```

Word 1 in this area:

```
UNITGROUPINFOFF = 39:20
% This is the number of words in the unit group
% info array.
UNITINDEX = 19:20
% This is the number of entries.
```

Words 2 through 255:

```
UNITGROUPFF = 39:20
```

```
UNITINDEX = 19:20

% This is the unit number.  These words are

% initialized to REAL (NOT FALSE).
```

OPTION INFORMATION

This area in JOBDESC is where the CONTROLLER OPTION info is saved as specified by the ""CO"" input message. There are only two options:

```
ERRORDP

MONITOR
```

When ERRORDP is set, a programdump is taken on any CONTROLLER error. When MONITOR is set, statements within CONTROLLER will be conditionally executed.

Example to set option:

```
CO + ERRORDP <etx>
```

Example to reset option:

```
CO ERRORDP <etx>
```

Word 0 in this area:

```
OPTIONHEAD = OPTIONLENZ % 60

      % OPTIONVALIDITY % 12

      VALIDITYF % 47:8
```

Word 1 in this area:

```
SAVEOPTIONS -?

      0:1 ERRORDP

      1:1 MONITORF
```

Word 2 in this area:

```
SAVEMIXLIMIT
```

HEADER IN THE JOBDESC FILE

The header in the JOBDESC file is the same as headers found in

the FLAT DIRECTORY with the following exceptions:

WORD 6: JOBORDER

47:43 JOBORDERF

WORD 9: DISKLINK, used to link jobs together in a job queue.

WORD 11: JOBNUMBER

47:1 JOBVALIDF

46:1 JFBIT, got a job file to print.

15:16 JOBNBF

WORD 17: BACKUPCONTROL

"JOB QUEUE" ODT MESSAGES

A brief description of ODT messages associated with the queues is given below:

EQ Eliminates a given queue and redistributes its entries. Typical response:

queue 123 eliminated

PQ Removes all jobs from a given queue (leaving the queue intact) but does not redistribute its entries. Typical response:

queue 5 purged

QF Displays a given queue's factors. Typical response:

QUEUE 0:

DEFAULT QUEUE

MIXLIMIT = 10

ACTIVE COUNT = 1

DEFAULTS:

PRIORITY = 50

LIMITS:

PRIORITY = 50

MQ Makes a new queue or modifies an existing queue.

TYPICAL INPUT:

```
MQ 3 MIXLIMIT = 3, TURNAROUND = 5.

DEFAULTS (PRIORITY = 50).

LIMITS (PRIORITY = 50) <etx>
```

SQ Shows jobs in the job queues.

TYPICAL RESPONSE:

```
QUEUE 7:

6627 80 ? JOB A;

6631 80 ? JOB D;

6629 00 ? JOB C;
```

DQ Sets designated queue as default queue.

MISCELLANEOUS CONTROLLER NOTES

CONTROLLER makes use of quite a lot of local procedures and variables. A few of the most important of these are discussed in the following paragraphs.

DISKMAP ARRAY

The "HEX" array DISKMAP, represented in Figure 9-12, is used by the CONTROLLER procedures GETDISK and GIVEDISK to handle JOBDESC file space allocation. Each entry in DISKMAP represents a disk segment and since entries may require more than one segment, a given job entry may require multiple entries in the DISKMAP. Entries in the DISKMAP may have the following values:

4"0" Available space.

4"1" This is a filler and indicates that this segment is part of a larger record.

4"2" This marks the start of a job entry.

4"3" This also marks the start of a job entry but also indicates that this job is in the print phase.

In reference to the procedure GETDISK, DISKMAP is searched for a contiguous group of 4"0" entries indicating that the

associated segments are available.

After an area has been found (large enough) the starting segment is marked with a 4"2" in the map. If more than one segment is required, then the remaining segments are marked with a 4"1".

The record in the JOBDESC file corresponding to the starting digit in DISKMAP will be updated to contain the size just allocated. This is necessary to preserve the map should a halt/load occur.

THE "ABSTRACT" PROCEDURE

The ABSTRACT procedure is passed a HDR/PPB and uses the information contained within (job attributes) to build another array in "WINDOW" format (see Figure 9-13). This newly constructed array is used by ABSTRACT to select the proper queue for the new job. ABSTRACT, itself, does not insert a job into a queue but does return the appropriate queue number.

The compile time option, QFACTMATCHING, if set will cause any job which has not specified a class (and has not specified any job attributes) to only enter the DEFAULTQ. If the option is set and options are specified, then a best fit will be attempted. If the option is reset and class is not specified, then the job will enter the DEFAULTQ regardless of the other job attributes. Figure 9-14 shows the array in which job attributes are stored. Note that this array is used to store information about a queue and not about any specific job in that queue as is done in the "WINDOWS" array.

THE "QUEUEINSERT" PROCEDURE

The QUEUEINSERT procedure is the CONTROLLER procedure used to directly insert jobs (HDR/PPB's) into the queues. This procedure is passed the queue number and an array in window format. Note here that the first word in a window formatted array is DISKLOCP, the index in segments into the JOBDESC file where the HDR/PPB has already been written.

If there are no entries in the queue or the priority of the given job is greater than the job at the head of the queue (in WINDOWS Figure 9-13) the new job will be placed in the WINDOW. The job is then linked into the "DISKQUEUE" (Figure 9-15) of all jobs with a similar priority for the given queue. For each queue, DISKQUEUES, contains the head and tail disk addresses for all job linked together by similar priority.

INDEX	HEX DIGITS IN "DISKMAP"	SEGMENTS IN "JOBDESC"
0	2	HEADER/PPB
1	2	HEADER/PPB
2	2	HEADER/PPB
3	1	
4	1	
5	1	
6	0	AVAILABLE
7	0	
7	0	
9	2	HEADER/PPB
10	1	
11	0	AVAILABLE
12	0	
13	3	HEADER/PPB
14	0	AVAILABLE

.
.
.

ENTRIES IN DISKMAP:

0 AVAILABLE
2 START OF A JOBDESC ENTRY.
1 FILLER, INDICATES A CONTINUATION OF THE ENTRY STARTED WITH THE PRECEEDING "2"
3 SAME AS "2" BUT LINKED INTO THE PRINT QUEUE.

DISKMAP WAS DECLARED AS FOLLOWS:

```
HEX ARRAY DISKMAP [0:CHUNKSIZE*CHUNKSPERROW*NUMROWS-1];
                    = 3        = 150      = 40
```

Figure 9-12. DISKMAP ARRAY

Figure 9-13. WINDOWS ARRAY.

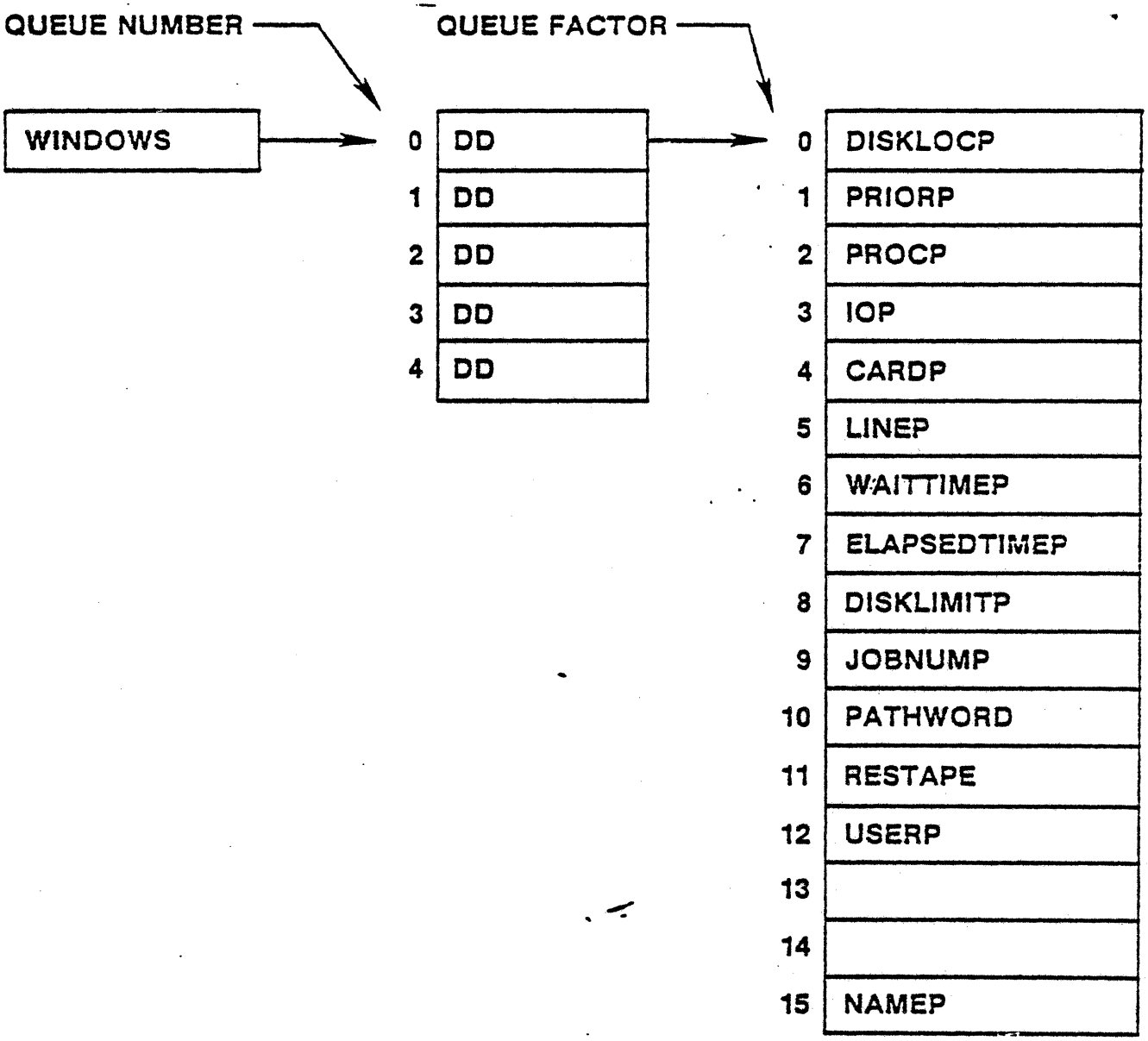


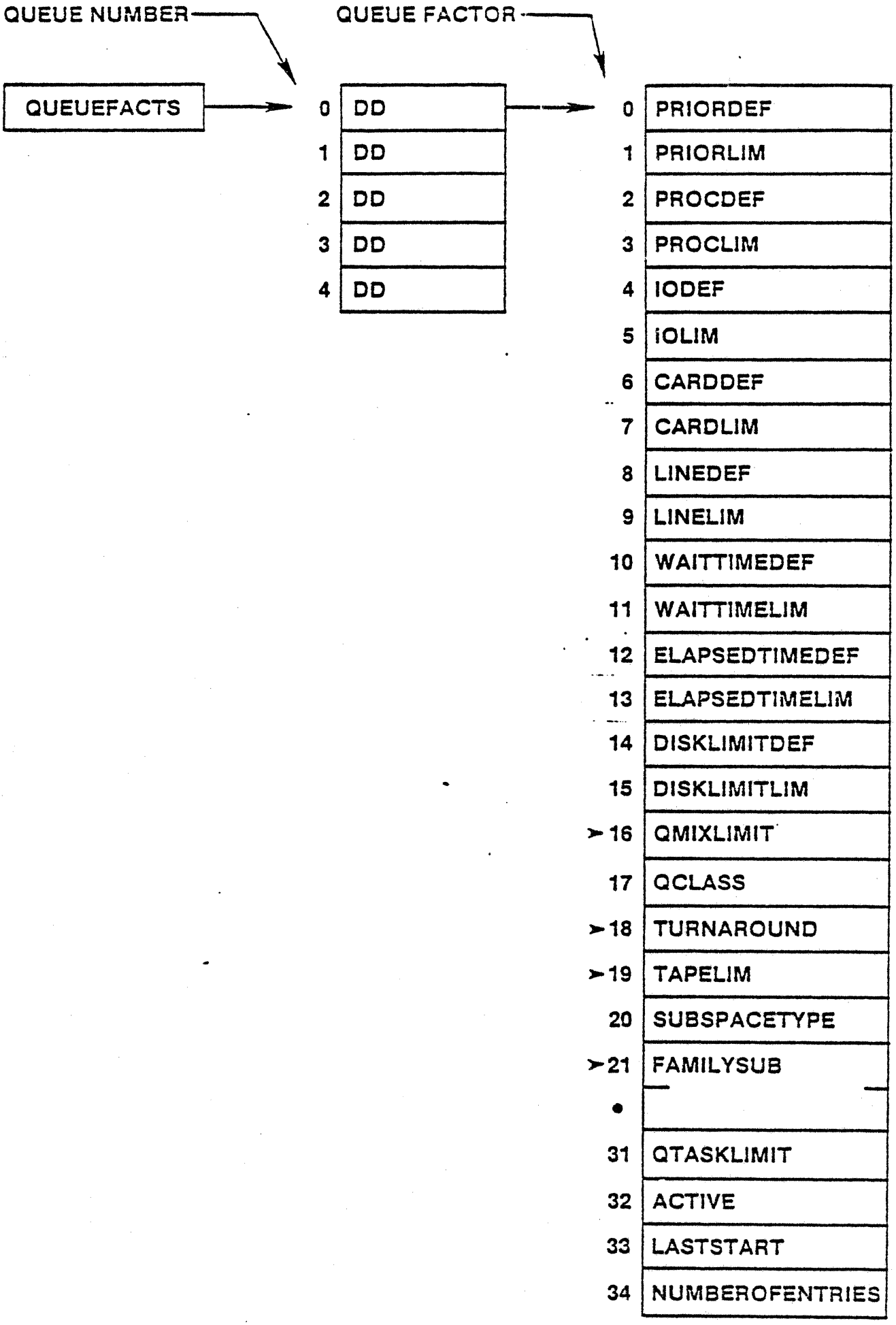
Figure 9-13. WINDOWS ARRAY.

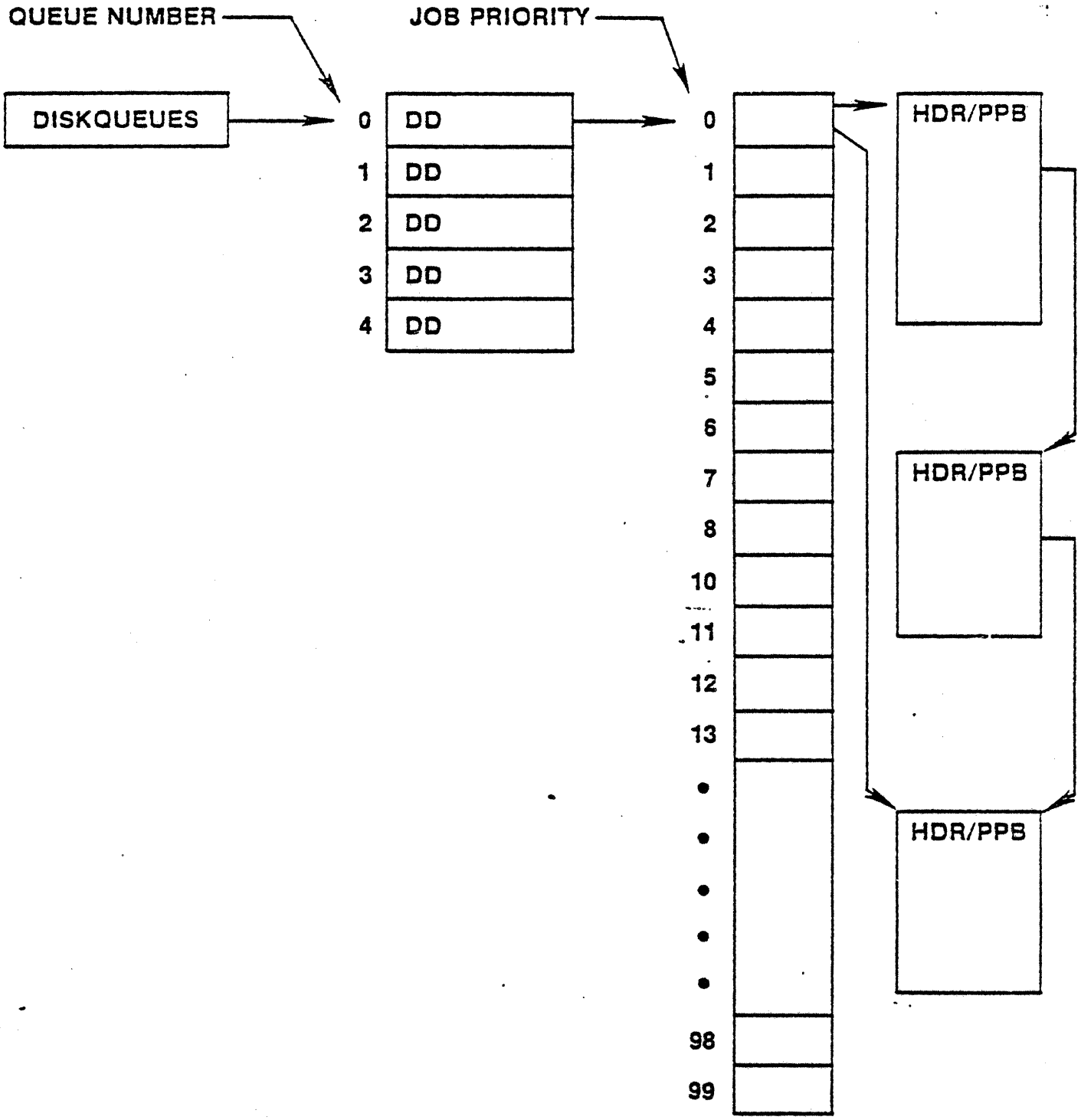
Figure 9-14. QUEUEFACTS ARRAY.

LAYOUT OF THE "PRIORITY" WORDS: QUEUETAILF = 47:24

QUEUEHEADF = 23:24

Figure 9-15. DISKQUEUES ARRAY.


QUEUE NUMBER
QUEUE FACTOR
QUEUEFACTS
0 DD
1 DD
2 DD
3 DD
4 DD
0 PRIORDEF
1 PRIORLIM
2 PROCDEF
3 PROCLIM
4 IODEF
5 IOLIM
6 CARDDEF
7 CARDLIM
8 LINEDEF
9 LINELIM
10 WAITTIMEDEF
11 WAITTIMELIM
12 ELAPSEDTIMEDEF
13 ELAPSEDTIMELIM
14 DISKLIMITDEF
15 DISKLIMITLIM
16 QMIXLIMIT
17 QCLASS
18 TURNAROUND
19 TAPELIM
20 SUBSPACETYPE
21 FAMILYSUB
31 QTASKLIMIT
32 ACTIVE
33 LASTSTART
34 NUMBEROFENTRIES



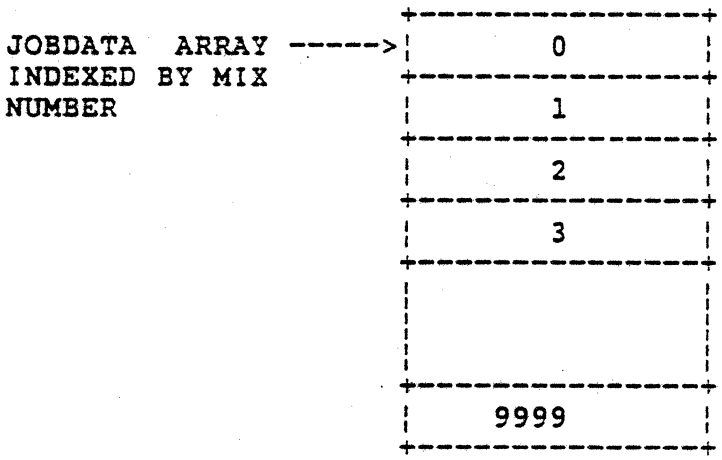
LAYOUT OF THE "PRIORITY" WORDS: QUEUETAILF = 47:24

QUEUEHEADF = 23:24

Figure 9-15. DISKQUEUES ARRAY.

JOBDATA ARRAY

The JOBDATA array, Figure 9-16, is and MCP declared array that is used throughout CONTROLLER. The most important field in the words of this array is LOCATIONF. This field contains the same information as DISKLOCP (described earlier). The array is indexed by job number. At end-of-job time, CONTROLLER is passed the job number (by the MCP) which he uses to index the JOBDATA array and, using the LOCATIONF, finds the HDR/PPB in the JOBDESC file. Note that the HDR/PPB was in the JOBDESC file during the entire execution of the job.



LAYOUT OF JOBDATA WORDS:

JOBVALIDF	47:1
OFFSPRUNGF	46:1
STKNRF	45:10
MCSONLYF	35:1
PRIORITYF	34:7
INQUEUEF	27:1
CPOFFSPRUNGF	26:1
JOBFILEPRINTEDF	25:1
QUEUEF	24:5
LOCATIONF	19:20

Figure 9-16. JOBDATA ARRAY

CONTROLLER WHILE WAITING

CONTROLLER is normally found waiting on any of 5 events as indicated in figure 9-17. One of the events, ADMTIMEOUT, is a real procedure and produces a wait time. The other 4 events are located in the HIDDEN MESSAGE of the associated queues.

Figure 9-18 was included here to give a detailed example of a typical CONTROLLER process stack during the wait period. Note that the events are indexed data descriptors. The MCP procedure, SUPERWAIT, is always called when more than one event is to be waited on to provide an interface to MULTIPLEWAIT This is required because of the variable number of parameters that may be passed. SUPERWAIT and DELIVERY serve identical functions but where DELIVERY always calls INITIATEUSERTASK, SUPERWAIT always calls MULTIPLEWAIT.

```
WHILE I := WAIT ((ADMTIMEOUT).
OPERATORINPUTQ.QINSERTEVENT.

CONTROLLERQ.QINSERTEVENT.

REMOTEINPUTQ.QINSERTEVENT.

MESSAGEDISPLAYQ.QINSERTEVENT) IS 1 DO;
```

```
INSTRUCTION                    COMMENT
-----------                    -------

MKST
NAMC 0,9F          SUPERWAIT PROCEDURE IN THE MCP.
MKST
NAMEC 2A8          ADMTIMEOUT, A REAL PROCEDURE WITHIN CONTROLLER.
ENTR               ENTER THE ADMTIMEOUT PROCEDURE.
ZERO
LT8                HIDDEN MESSAGE INDEX FOR QINSERTEVENT.
NAMC 1,2           OPERATORINPUTQ
STFF
LOAD
INDX
XTND
ZERO
LT8  8             HIDDEN MESSAGE INDEX FOR QINSERTEVENT.
NAMC 1,3
STFF
LOAD
INDX
XTND
ZERO
LT8  8           HIDDEN MESSAGE INDEX FOR QINSERTEVENT.
NAMC 2,D         REMOTEINPUTQ
STFF
LOAD
INDX
XTND
ZERO
LT8  8           HIDDEN MESSAGE INDEX FOR QINSERTEVENT.
NAMC 1,4         MESSAGEDISPLAYQ
STFF
LOAD
INDX
XTND
ZERO
ENTR               ENTER THE SUPERWAIT PROCEDURE.
NAMC 2,45          I, A VARIABLE DECLARED IN CONTROLLER.
STON
ONE
SAME
BRFL             BRANCH OUT OF THIS LOOP AND INTO THE NEXT
                 STATEMENT IF ""I"" IS EQUAL TO 1.
BRUN             BRANCH BACK TO THE TOP MKST INSTRUCTION TO CONTINUE
                 THE LOOP.
```

Figure 9-17. WHAT CONTROLLER IS WAITING ON

```
0 000000 000035
3 2187A6 F88005
3 C1A001 E04009     STACKMOVER
0 000000 000000
0 000000 000000
0 000000 000000
5 C0004F D047A5
0 000000 000000
0 000000 000035
0 000000 000000
3 230416 F84006
3 C08475 708008     NORMALGEORGE
0 000000 000001
0 000000 0048F3
0 000000 0048FB
5 E10000 805699
0 000000 0048FB
0 000000 0048EF
3 218618 184006
3 C1A001 E0400C
0 000000 000000
5 E10000 805699     TO MESSAGEDISPLAYQ
0 000000 000000
5 E10000 8056A8     TO REMOTEINPUTQ (DECL. IN CONTROLLER)
0 000000 000000
5 E10000 804237     TO CONTROLLERQ
0 000000 000000
5 E10000 80568A     TO OPERATORINPUTQ
0 034065 B7B620
0 020240 C03E35     BITS 39:20 POINT TO "CALENDAR"
3 21840A C087A1
3 C1A001 E04115     SUPERWAIT
6 800000 013800
3 218403 1045A1
3 434002 D08008     CONTROLLER
6 800000 002800
5 800000 33D86C
5 800003 505084
1 41A001 E02000
7 034200 2085A2
0 000000 000020
3 000000 0883FE
3 C1A001 E04002     CONTROLLOR
3 000000 002000
3 C08475 70802A     NORMALEOJ
0 000000 000000


3 434000 00001A     STUFFITMSCW
3 200016 984002     TOSCW
```

Figure 9-18. TYPICAL CONTROLLER STACK

AUTOBACKUP

Information to be written to a printer or punched in cards is usually written to backup disk instead. The information may now be printed or punched at a more convenient time. There are other advantages as well.

In reference to figure 1-6, the MCP by use of the CONTROLLERQ, notifies CONTROLLER that a job has finished and to output the JOB SUMMARY, WORK FLOW STATEMENTS, and backup for the terminating job. CONTROLLER does not do any of this work so he passes the job on to AUTOBACKUP. This is accomplished by calling BACKUPQUEUER, an MCP procedure, to place an entry in the AUTO PRINT QUEUE. If possible, AUTOBACKUP is forked and it is this procedure (IR) along with JOBFORMATTER that actually does the printing or punching. It is important to note that JOBFORMATTER is only responsible for printing the JOB SUMMARY and WORK FLOW STATEMENTS (both obtained from the JOBFILE). AUTOBACKUP does the printing or punching of the real backup files.

For a more detailed discussion of AUTOBACKUP, refer to figure 9-19. When a job ends, NORMALEOJ is exited into and this procedure, seeing that it's a job that finished, calls DCINSERT to place an EOJNOTICE in the CONTROLLERQ. CONTROLLER now executing, responds to the EOJNOTICE (known as EOTNOTICE in CONTROLLER) by calling its local procedure, PRINTJOB, which in turn, calls the MCP procedure BACKUPQUEUER passing an ENQUEUEV code and the mix number of the terminating job. BACKUPQUEUER places an entry in the AUTO PRINT QUEUE and then exits back to PRINTJOB. CONTROLLER will continue by going back to sleep on its 5 events.

If BACKUPQUEUER had determined that an auto-printer could be started, then AUTOBACKUP would have been forked and passed the entry that had been placed in the AUTO PRINT QUEUE.

AUTOBACKUP works on one job at a time. First, JOBFORMATTERHOST is called and passed the job number. This procedure calls JOBFORMATTER to print the JOB SUMMARY and WORK FLOW STATEMENTS. After this is done, an exit is made back into AUTOBACKUP and AUTOBACKUP continues by printing or punching all of the job's backup files. PRINTLIST is the procedure local to AUTOBACKUP that does this.

Each time AUTOBACKUP finishes a job, an EOBNOTICE (end of backup) is sent to CONTROLLER and AUTOBACKUP continues by calling BACKUPQUEUER passing a DEQUEUEV and the job number to be removed from the AUTO PRINT QUEUE. If there are no more entries in the queue or a CM or EI is active then BACKUPQUEUER returns a 0 which will cause AUTOBACKUP to leave the mix. If there are more entries in the queue (and no CM or EI) then BACKUPQUEUER will return the address of the next queue entry to AUTOBACKUP.

When CONTROLLER receives an EOBNOTICE, CONTROLLER will call FORGETUSERDISK on the JOBFILE's header. CONTROLLER will also call its local procedure, GIVEDISK. This procedure marks the JOBDESC location that was being used by the HDR/PPB (JOB QUEUE ENTRY), as available. This is done in CONTROLLER's DISKMAP array.

Figure 9-19. AUTOBACKUP ABSTRACT

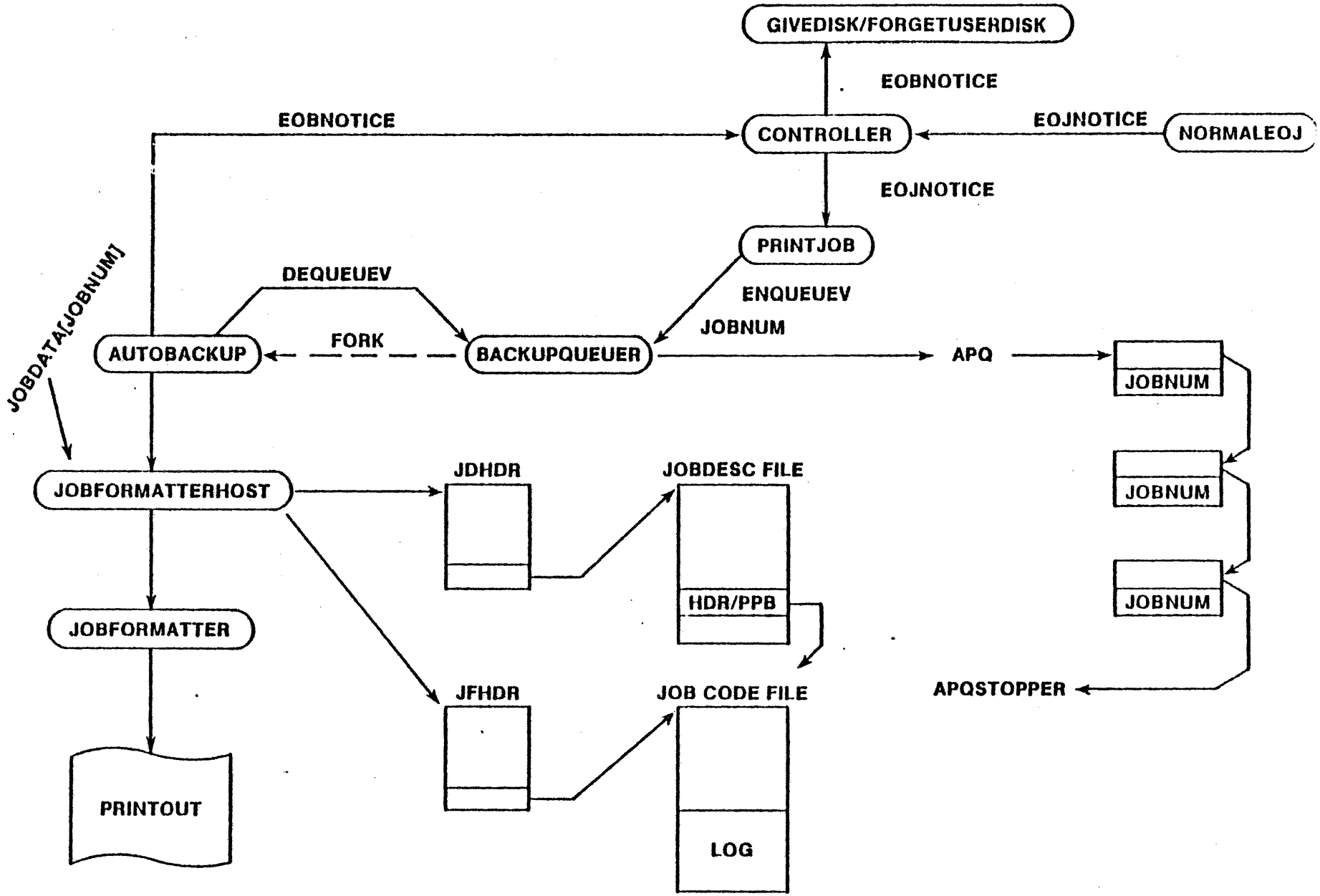


Figure 9-19. AUTOBACKUP ABSTRACT

MISCELLANEOUS PROCESS CONTROL PROCEDURES

This sub-section starts with a lengthy discussion on events and software interrupts. This is followed by a discussion on the MCP procedures TIMETUNNEL, WAITP, and CAUSEP, DELIVERY/INITIATEUSERTASK, etc. A few outlines are included but will not be discussed in the text.

EVENT AND INTERRUPT MECHANISM

General

During the course of Interprogram Communications, it becomes necessary to synchronize and/or provide interlocks between two or more tasks of the IPC environment. This is provided for via the EVENT and INTERRUPT mechanism. The purpose of this discussion is to provide a guide as to the concepts and use of this mechanism. Since this discussion is general in nature, it will be necessary for the reader to consult individual language manuals for syntax and semantic details.

The EVENT and INTERRUPT mechanism can be roughly divided into four broad discussion areas: EVENT DECLARATION, CONDITION-ORIENTED FUNCTIONS, RESOURCE-ORIENTED FUNCTIONS, and, finally, the INTERRUPT MECHANISM.

EVENT Declaration

The EVENT is a special type of variable declared in a manner similar to other types of variables, that is:

ALGOL and NEWP

```
EVENT E1
```

COBOL

```
77 E1 USAGE IS EVENT
```

As to other types of variables, the EVENT can be passed as parameter to other tasks, thus allowing more than one task to access the same EVENT.

The event itself has two properties, each property having two states. The condition-oriented property has a HAPPENED or NOT HAPPENED state, and the resource-oriented property has an

AVAILABLE or NOT AVAILABLE state. The initial state of each property is NOT HAPPENED and AVAILABLE. Associated with each property is a set of functions used to control and/or interrogate the state of the property.

CONDITION-ORIENTED FUNCTIONS.

One utilization of the EVENT is to notify one task of a condition detected or established by another task. For example, if one has two tasks, PGEN which generates data into an array shared with a task PUSE which extracts and uses the data, it is necessary for PGEN to notify PUSE when the array contains valid data. It is also necessary for PUSE to notify PGEN when it has extracted the data so that PGEN may refill the array. Syntax for the situation- oriented functions is as follows:

a. SET FUNCTION:

(1) ALGOL

SET (<event-ID>)

(2) COBOL

Not implemented

(3) NEWP

SETEVENT (<event-ID>)

The SET function will set an event to the HAPPENED state. It will not cause any other action, i.e. the SET function will not activate a task waiting on the event.

b. RESET FUNCTION;

(1) ALGOL

RESET (<event-ID>)

(2) COBOL

RESET <event-ID-1>[,<event-ID-2>]...

(3) NEWP

RESETEVENT (<event-ID>)

The RESET function will reset an event to the NOT HAPPENED state. It will not cause any other action.

c. CAUSE FUNCTION:

(1) ALGOL NEWP

CAUSE (<event-ID>)

(2) COBOL

CAUSE <event-ID-1>[,event-ID-2>]...

The CAUSE function will activate all tasks which had been waiting on the event. Normally the cause will also set the event to the HAPPENED state. (See the WAIT AND RESET function for exceptions.)

It must be pointed out that activating a task does not imply the task goes into immediate execution. Activating a task consists of delinking the task from an event queue (each event has its own queue) and linking that task in priority order into a system queue called the READYQ. The READYQ is a queue of all tasks, in, priority order, that are capable of running. Tasks are taken out of the READYQ when either a processor is assigned to the task, or the task must wait for something such as an I/O operation or an event to be cause. A task will only be placed into actual execution when it is the top item in the READYQ and a processor is available.

d. CAUSE AND RESET FUNCTION:

(1) ALGOL NEWP

CAUSEANDRESET (<event-ID>);

(2) COBOL

Not implemented

The CAUSE-AND-RESET function is similar to the cause function in that it activates all tasks waiting on the event. It varies from the cause function in that the resultant state of the event is set to NOT HAPPENED.

e. SIMPLE WAIT FUNCTION:

(1) ALGOL COBOL NEWP

WAIT (<time>)

WAIT (<event-ID>)

<time> ::= <the amount of time in seconds (fractional seconds allowed) that a task is to be suspended>

The SIMPLE WAIT function allows for suspending a task either for a time period or until an event is caused.

For the WAIT-ON-TIME syntax, the time function generates an implicit event which it waits on. This event is caused by the operating system when it detects the time period specified by <time> has elapsed. It might be noted that depending on the degree of multiprocessing being performed and program priorities, the actual time a task is suspended or <time> may vary widely in respect to the time indicated in <time> with smaller increments of time having the greatest variation.

For the WAIT-ON-EVENT syntax, the function examines the happened state of the event. If the event state is HAPPENED, the function is essentially a no-operation. If the event state is NOT HAPPENED, the task will be suspended until such time some other task executes a CAUSE statement.

f. SIMPLE WAIT AND RESET FUNCTION:

(1) ALGOL NEWP

WAITANDRESET (<event-ID>)

(2) COBOL

Not implemented

The WAIT AND RESET function allows for suspending a task until the event is caused. it is identical to the simple wait except that the event is forced to the state NOT HAPPENED during the subsequent cause processes.

g. TERMINATE WAIT FUNCTION:

(1) NEWP

WAIT [DSABLE] (<event-ID>)

(2) ALGOL COBOL

Not applicable

There exist some situations whereby certain operating system functions cannot be terminated while waiting on an event, such as while waiting for an I/O complete, etc. For this reason, a request to terminate a system function which is in turn waiting on an event will not be honored. However, there are other operating system functions for which no system problem occurs if they are terminated while waiting on event. For these functions, the DSABLE option is used.

Termination of operating system functions operates somewhat differently than for object tasks. First of all an interlock is set so that control cannot return to the object program which called the system function. The system function then is activated as if the event has been caused. Means exist whereby a system function can determine if it was activated by

a cause of an event or by a terminate request.

h. COMPLEX WAIT FUNCTION:

(1) COBOL

Not implemented

(2) ALGOL NEWP

WAIT (<wait-parameter-list>)

WAITANDRESET (<wait-parameter-list>)

<wait-parameter-list> ::= <aexp>,

<event-list> ¦ <event-ID>,<event-list>

<aexp> ::= <time period in seconds>

<event-list> ::= <event-ID>, <event-list> ¦ <event-ID>

The COMPLEX WAIT function allows a task to be suspended until any one event in the <event-list> is caused or until the time as indicated by <aexp> (in seconds) has elapsed.

The COMPLEX WAIT function is an integer function which returns a value (starting at 1) which represents the position in the <wait-parameter-list> of the parameter which caused the task to be activated. For example, in the statement

T ::= WAIT (.001, E1, E2)

the value of T would be 1 if elapsed time caused the task to be activated, while in the example

T ::= WAIT (E1, E2, E3)

the value of T would be 2 if a cause on event E2 activated the task.

It might be noted that the implementation of this mechanism contains interlocks to guarantee that one and only one parameter can activate a task.

If time is included as a parameter, it must be the first parameter in the list.

WAIT and WAITANDRESET are identical except for the state to which the caused event is set during the cause process. If all tasks are waiting on the event via the WAIT statement (simple or complex), the state of the event is set to HAPPENED. If any one task is waiting on the event via the WAITANDRESET statement, the state of the event is set to NOT

HAPPENED.

At times it will be found that a particular function can be implemented either by utilizing the complex wait mechanism or the interrupt mechanism. It is recommended that when this trade off occurs, the complex wait mechanism be used as the system overhead in handling the complex wait mechanism is substantially less than it is for the interrupt mechanism.

i. HAPPENED TEST:

(1) ALGOL NEWP

IF HAPPENED (<event-ID>) THEN <statement>

(2) COBOL

IF <event-ID> <statement>

The HAPPENED test provides a means to test the state of the condition-oriented property of an event. The test returns a true condition if the event is in the HAPPENED state and a false if the event is in a NOT HAPPENED state.

One must be extremely careful in using the happened test, as system inefficiencies and program failure can occur. By way of explanation, consider the following:

```
WHILE NOT HAPPENED (E1) DO

    <statement>
```

where <statement> may be <empty> or of such a type that explicit (WAIT) or implicit (I/O) statements which imply temporary task suspension are NOT done.

This loop causes two problems. First of all, it will tie up a processor completely. One should always use the WAIT function when a point is reached where processing cannot proceed until an EVENT is caused. The WAIT function releases the processor to some other task allowing that task to do useful work. The second problem occurs when only one processor is available, either due to the system being a one-processor installation or one processor is undergoing repair. If the loop is in a high priority task and the CAUSE (or SET) statement is in a lower priority task, the situation exists where the higher priority task utilizes all available processor time executing the loop. The result is that the lower priority task cannot execute the code which changes the state of the event, causing the higher priority task to loop endlessly.

It might be noted that the above should be considered not only when using the HAPPENED test but also when using the AVAILABLE test (see resource-oriented properties of events), the readlock mechanism or when using shared variables between

tasks for inter-task control.

Some General Comments

By this time the reader may be concerned as to setting of the HAPPENED state of an event just after an event is caused. Essentially any one task waiting on the event via the WAITANDRESET function or the event control task executing the CAUSEANDRESET statement will result in the state NOT HAPPENED. All tasks must be waiting using the WAIT function and the event control task must use the CAUSE function in order for the event to have the HAPPENED state immediately after the cause process. For this reason, extreme care must be taken when using these functions. For this reason, extreme care must be taken when using these functions. As a guide to the proper use of the WAIT, WAITANDRESET, CAUSE, and CAUSEANDRESET functions, consider there are two types of conditions, momentary and elapsed.

A momentary condition can be defined as a condition which can exist for only an instant. The CAUSEANDRESET and WAITANDRESET functions allow the implementation of momentary conditions. For the situation where event control task (the one causing the event) solely determines the condition, the dependent stacks (those waiting) should use the WAIT function while the event control stack should use the CAUSEANDRESET function.

For the case where the act of a task being activated means the condition is to disappear, the dependent task should use the WAITANDRESET function while the event control task should use the CAUSE function. One should avoid mixing WAIT, WAITANDRESET, CAUSE and CAUSEANDRESET all on the same event. the resultant confusion over the HAPPENED state of the event can cause a considerable problem.

An elapsed condition can be defined as a condition which holds over a long period. To implement this type of condition, one should use only the WAIT, CAUSE, and RESET functions.

RESOURCE-ORIENTED FUNCTIONS.

The second property of an event is the resource-oriented property. One can consider a resource as something which can be utilized by several tasks but only one task at a time. For example, let us consider a complex list structure built into an array as a resource. Let us also assume that there exists a task which adds data to the list structure and another task which deleted data from the list structure. We also assume that both tasks running simultaneously would destroy the list structure. It is necessary to let either task run at any time but not simultaneously. This type of control can be done via

the resource-oriented properties of an event and the RESOURCE-ORIENTED functions, which are as follows:

a. UNCONDITIONAL PROCURE FUNCTION:

(1) ALGOL NEWP

PROCURE (<event-ID>)

(2) COBOL

LOCK (<event-ID>)

Note: NO ""AT LOCKED"" CLAUSE

The UNCONDITIONAL PROCURE function tests the available state of an event. If the event is NOT AVAILABLE, the task executes the WAIT function. If the event was AVAILABLE, the event state is set to NOT AVAILABLE and the task continues in sequence.

b. CONDITIONAL PROCURE FUNCTION:

(1) ALGOL NEWP

FIX (<event-ID>)

(2) COBOL

LOCK (event-ID) AT LOCKED <statement>

The CONDITIONAL PROCURE function is a Boolean function which examines the available state of an event. If the state is AVAILABLE, the event is procured (sets state to NOT AVAILABLE) and a false is returned. If the available state was NOT AVAILABLE, the function returns a true, leaving the available state unchanged, COBOL uses the truth value returned, implicitly, to control the branching associated with the ""AT LOCKED"" clause.

One should be careful in using the CONDITIONAL RESOURCE function to avoid the loop problem mentioned previously in the HAPPENED test.

c. LIBERATE FUNCTION:

(1) ALGOL NEWP

LIBERATE (<event-ID>)

(2) COBOL

UNLOCK (<event-ID>)

The LIBERATE function, when executed, produces several

effects.

First of all, the procure list is examined. If there are no other tasks waiting to procure the event, the EVENT state is set to AVAILABLE. If there are other tasks waiting to procure the event, the highest priority task is activated. The EVENT state is left marked as NOT AVAILABLE.

Lastly, all tasks waiting on the event are activated (an implicit cause is executed). This may result in a change to the HAPPENED state of the event depending on whether the tasks which were waiting used the WAIT or the WAITANDRESET function.

d. FREE FUNCTION.

(1) ALGOL NEWP

FREE (<event-ID>)

(2) COBOL

Not implemented

This function is a Boolean function which examines the availability state of an event. It returns a true if the event is AVAILABLE and a false if the event is NOT AVAILABLE. In addition, it will reset the EVENT state unconditionally to AVAILABLE but will not activate any task suspended by an attempt to procure the event nor will it activate any task waiting on the event.

e. THE AVAILABLE TEST:

(1) ALGOL NEWP

IF AVAILABLE (<event-ID>)

(2) COBOL

Not implemented

The AVAILABLE test is a Boolean function which examines the available state of an event. It returns a true if the event is AVAILABLE and a false if the event is NOT AVAILABLE.

Caution should be observed in using the AVAILABLE test to avoid the loop problem mentioned previously in the discussion under the HAPPENED test.

THE INTERRUPT MECHANISM.

An interrupt is a body of code which can be associated with an event. When the event is caused, processing of the main

program can be interrupted and the interrupted code will be executed. At completion of execution of the interrupt code, control will return to the main program at the point the main program was interrupted unless the interrupt code explicitly transfers control via a GO TO a main program label. Associated with the interrupt mechanism are several interrupt functions:

a. INTERRUPT DECLARATION:

(1) ALGOL NEWP

```
INTERRUPT <interrupt-ID>; <statement>
```

(2) COBOL (declarative section)

```
<interrupt-ID> SECTION.

USE AS INTERRUPT PROCEDURE.

<paragraph-name>.

(interrupt code goes here)
```

The INTERRUPT DECLARATION serves the purpose of naming particular interrupt code.

b. ATTACH FUNCTION:

(1) ALGOL NEWP COBOL

```
ATTACH <interrupt-ID> TO <event-ID>
```

The ATTACH function associates an interrupt with an event. The association is such that a cause of the event will interrupt the main program and place the interrupt code into execution, providing the interrupt has been allowed (see ALLOW, DISALLOW).

While different interrupts can be simultaneously attached to the same event, a particular interrupt can at any time be attached to only a single event. For this reason, if, at attach time, it is found that the interrupt is already attached to an event, it is automatically detached from the old event and then attached to the new event. Any pending invocations of the interrupt (see ALLOW, DISALLOW) are lost.

It is possible to attach an interrupt to an event which is declared in a different task (attach of a local interrupt to a formal event). This may lead to certain compiler-time or run-time (""UP LEVEL ATTACH"") errors if it is found potentially possible for the task containing the event to be terminated prior to termination of the task containing the interrupt.

c. DETACH FUNCTION:

(1) ALGOL NEWP

```
DETACH <interrupt-ID>
```

(2) COBOL

```
DETACH <interrupt-ID-1>[,<interrupt-ID-2>]...
```

The DETACH function breaks the association of an interrupt to an event. Any pending invocations of the interrupt (see ALLOW, DISALLOW) are lost.

Detaching an interrupt which is not attached is essentially a no-operation, i.e. no error mechanism invocation occurs.

d. ALLOW-DISALLOW FUNCTION:

(1) ALGOL NEWP

```
ENABLE <interrupt-ID>

DISABLE <interrupt-ID>
```

(2) COBOL

```
ALLOW <interrupt-ID=1>[,<interrupt-ID-2>]...

DISALLOW <interrupt-ID-1> <interrupt-ID-2>]...
```

Like the event, an interrupt has two states: ENABLED and DISABLED (NOT ENABLED). For the case where the interrupt state is DISABLED, a CAUSE performed on the event to which the interrupt is attached will cause the interrupt to be queued but NOT executed.

The ALLOW function sets the interrupt state to ENABLED. Execution of this function will cause all queued interrupts to be executed (and also be removed from this interrupt queue) and, in addition, will also cause immediate execution of subsequent interrupts when the associated event is caused.

The purpose of queuing interrupts is to guarantee the interrupts are not lost during the time they are attached. However, since the queuing of interrupts is expensive from a system overhead standpoint, one should operate with interrupts allowed (enabled) whenever possible.

The primary purpose of the ALLOW-DISALLOW function is to prevent conflicts. If the main program is about to modify a set of variables and it is know that an interrupts can occur, which may also modify these same variables, the main program can delay the execution of the interrupt by placing a DISALLOW in front and an ALLOW behind that set of main program code

which modifies the variables.

As previously stated, an interrupt has two states, ENABLED and DISABLED. Two points need to be emphasized concerning the interrupt's state. First of all, for all languages the initial state of an interrupt is ENABLED. An exception to this is in NEWP where the interrupt code is written as part of a SAVE procedure. For this situation only, the initial state of the interrupt is DISABLED. In addition, even though the interrupt code is part of a NEWP SAVE procedure, the process of enabling the interrupt marks the code as non-save, i.e., it will run in interruptable mode. Another point to be emphasized is that the ATTACH and DETACH functions do not change the state of an interrupt or, conversely, the state of an interrupt can be changed at any time without regard to whether the interrupt is attached or not. For example, consider the following ALGOL sequence:

```
(1)  ATTACH I1 to E1;

(2)  DISABLE I1;

(3)  DETACH I1;

(4)  ATTACH I1 TO E2;
```

At statement (1), I1 is enabled as that is its initial state, and at statement (4), I1 is disabled because the last state change (statement (2)) set it to disabled.

e. GENERALIZED ENABLE, DISABLE:

```
(1)  ALGOL NEWP

     ENABLED;

     DISABLE;

(2)  COBOL

     Not implemented
```

The GENERAL DISABLE function sets a flag associated with the task containing the GENERAL DISABLE statement which causes the system not to look for interrupt code to execute for this task. The effect of this is as if all interrupts for the task had been set to their DISABLED state. During this period, all interrupts whose associated events were caused are placed in an interrupt queue.

The GENERAL ENABLED function resets this flag thereby causing the system to look for and place into execution all enabled interrupts which are in the interrupt queue of this task.

It might be noted that previously DISABLED interrupts can be

ALLOWED (ENABLED) during the time a task is in a general disable state and these interrupts will be executed when the flag is reset by the "GENERAL ENABLE" function, providing, of course, the associated event has been caused.

f. THE WAIT FOR INTERRUPT FUNCTION:

(1) ALGOL

```
WAIT
```

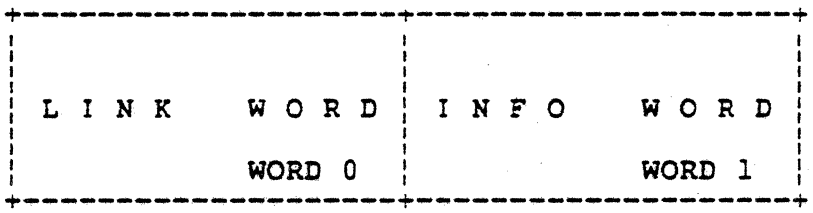
(2) NEWP

```
HOLD
```

(3) COBOL

```
WAIT INTERRUPT
```

This function allows a task to be suspended until such time as any one of the task's interrupts is placed into execution. At completion of the interrupt code execution, control can only be returned via an explicit GO TO from the interrupt code to a label within the main program.




```
LINK WORD LAYOUT
   [47:08] = CALTYPEF
         1 = CAUSESWAPOUT
         2 = CAUSEWAKEUP
   [39:20] = QBACKF
   [19:20] = QNEXTF

INFO WORD LAYOUT
   [45:10] = CALSNRF
   [35:32] = CALTIMEF
```

Figure 9-20. TIMETUNNEL ENTRY

TIMETUNNEL

TIMETUNNEL is the MCP procedure called anytime a process wishes to wait so many seconds before continuing. ONESECONDBURDEN (called by GEORGE) is the procedure that awakens the processes when their time is up.

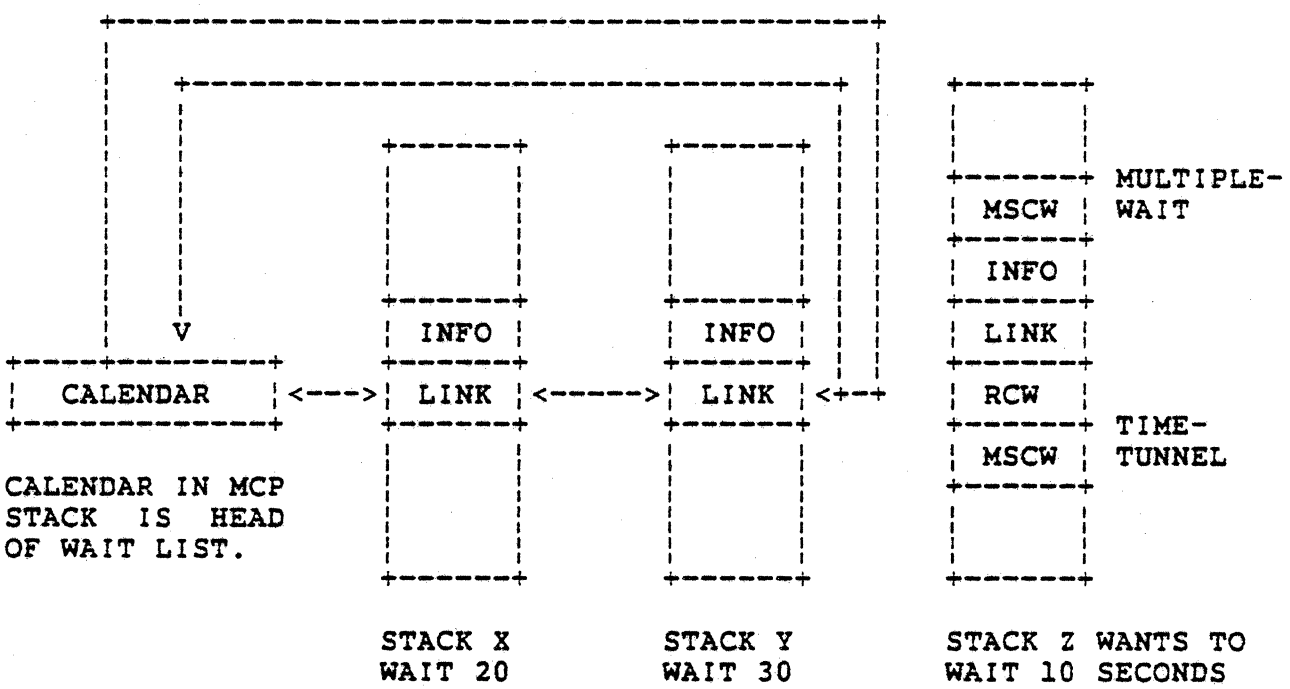
When TIMETUNNEL is called, a two word entry is made in the CALENDAR list. Entries are linked into this list in ascending order with the stack with the least amount of wait time first. The head of this list is the global variable CALENDAR and it is here that GEORGE looks (through the call on ONESECONDBURDEN) at approximately one second intervals. Starting at CALENDAR, ONESECONDBURDEN goes through the list resurrecting each stack until an entry is checked that has not met the time limit. Each time an entry is placed in the READY QUEUE (resurrected) it is delinked from the CALENDAR list.

Figure 9-20 is a TIMETUNNEL entry. This entry is part of TIMETUNNEL's portion of the process stack and may also be seen in figure 9-21. (If this stack is a swapjob stack the entry will be outside of the stack in an area obtained by GETAREA).

The first word in a TIMETUNNEL entry is a link word and the second word contains the stack number associated with the entry and the time the stack should be awakened. Figure 9-21 shows a TIMETUNNEL containing two entries and a third entry to be linked into the list. In reference to this figure, TIMETUNNEL links the stack into the list, makes BEDWORD, in the PIB, point to the MSCW immediately below the entry and then calls GEORGE. Figure 9-22 shows ""stack Z"" after it has been linked into the list. Notice the order of the wait times.

Assuming that about 11 seconds have passed and GEORGE has been called, GEORGE calls ONESECONDBURDEN. This procedure, using CALENDAR, find the first entry and, seeing that the time has come, delinks the entry from the tunnel and places it in the READY QUEUE. The next entry is checked but left alone since its time is not up yet.

The BEDWORD mentioned above is used to run through event lists awakening processes waiting on the events. This word will be discussed in the CAUSEP topic to follow.



TIMETUNNEL CALLS MULTIPLEWAIT. MULTIPLEWAIT BUILDS LINKAGE AND MAKES THE BEDWORD POINT TO THE FIRST EVENT THE STACK IS WAITING FOR. IN THIS CASE IT IS THE EVENT TIME.

Figure 9-21. TIMETUNNEL

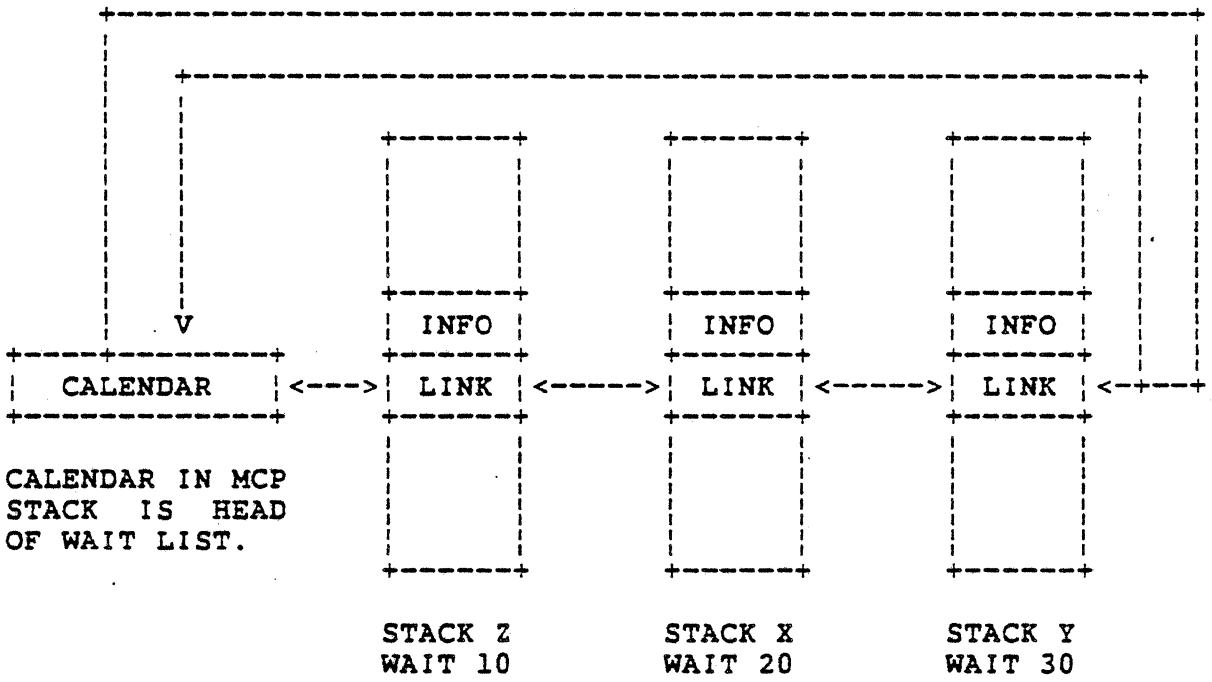


Figure 9-22. TIMETUNNEL

WAITP AND CAUSEP

WAITP is the MCP procedure called anytime a process wants to wait on only 1 event. WAITP is passed 2 parameters, an SIRW or indexed data descriptor to the event and a second parameter which simply indicates whether to set or reset the happened bit when the event is caused.

If the event passed to WAITP has already happened, then WAITP essentially becomes a NOOP and an exit is immediately made back into the calling procedure. If the event has not happened, WAITP links the event into the list of processes already waiting on the event, stores the absolute address of the WAITP MSCW in the BEDWORD of the PIB WAITP was called in and then calls GEORGE. GEORGE will call STACKMOVER and another stack will be moved to. This action can be seen in figures 9-23 and 9-24.

In figure 9-23 is an event in stack A. Stack B is waiting on this event. In figure 9-24, there are 3 stacks waiting on the same event. Notice how the 2 WAITP parameters are used to link the stacks into the list.

When an event is caused , the MCP procedure, CAUSEP, is called and passed to parameters in a fashion similar to that of WAITP. As with WAITP, the first parameter is an SIRW to the event and the second parameter is an indication of what to do after causing the event (set or reset the happened bit).

CAUSEP action is simple. In reference to figure 9-24, assume CAUSEP was called on the event shown in stack A. Seeing stack D is waiting on the event, CAUSEP places D in the READY QUEUE. Next, CAUSEP fetches stack D's BEDWORD in order to locate the link word at WAITP's MSCW +3. Since this word is not 0, CAUSEP places this stack number, C, in the READY QUEUE and continues in a similar manner with stack B. Now, with all stacks in the READY QUEUE, CAUSEP exits back into the calling procedure.

CAUSEP is also called from LIBERATEP. In this case it will wake up all stacks waiting on the event. In addition, the first stack wanting to PROCURE the event will wake up. It will also have control of the event.

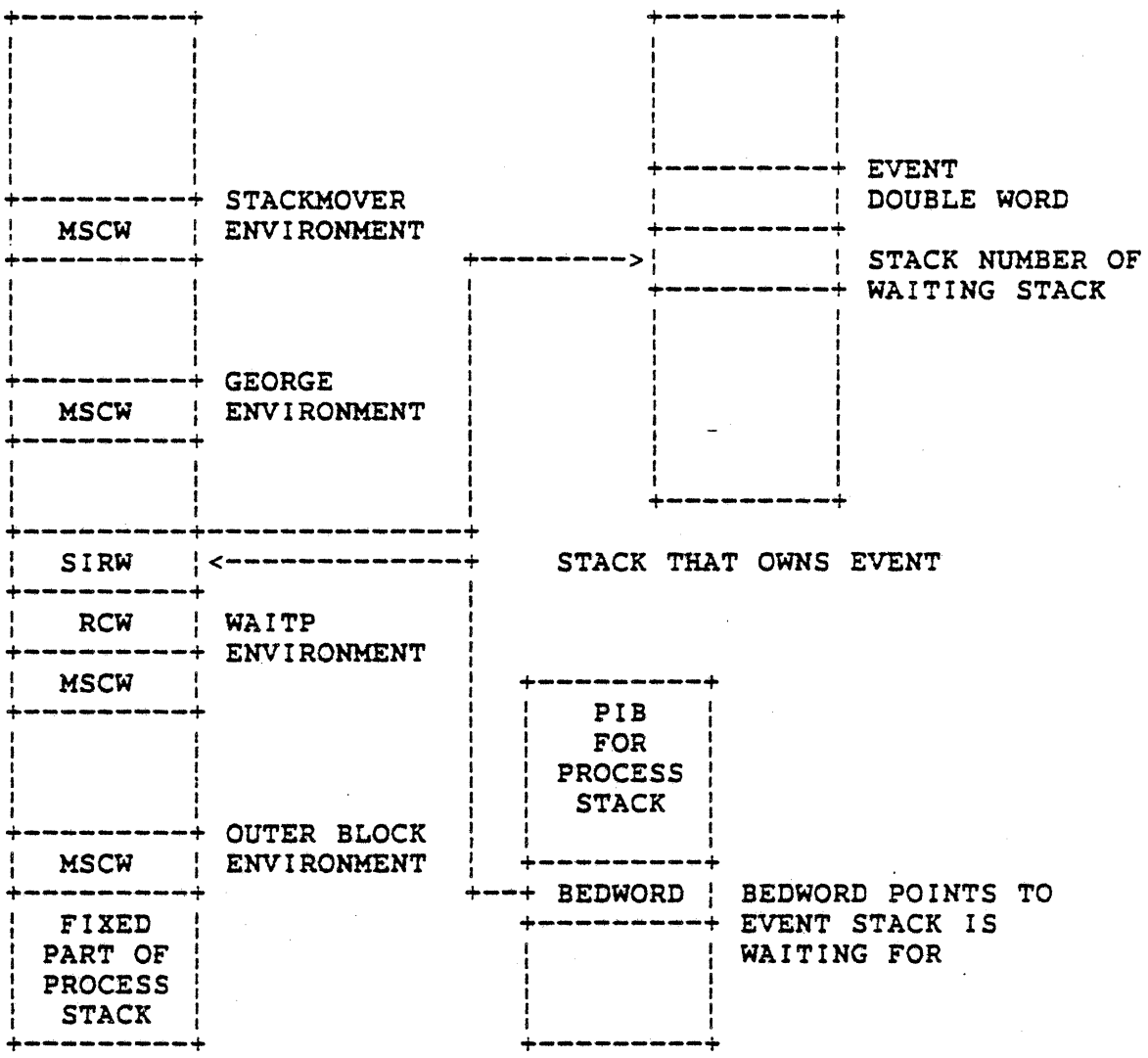


Figure 9-23. SINGLE STACK WAITING ON ONE EVENT

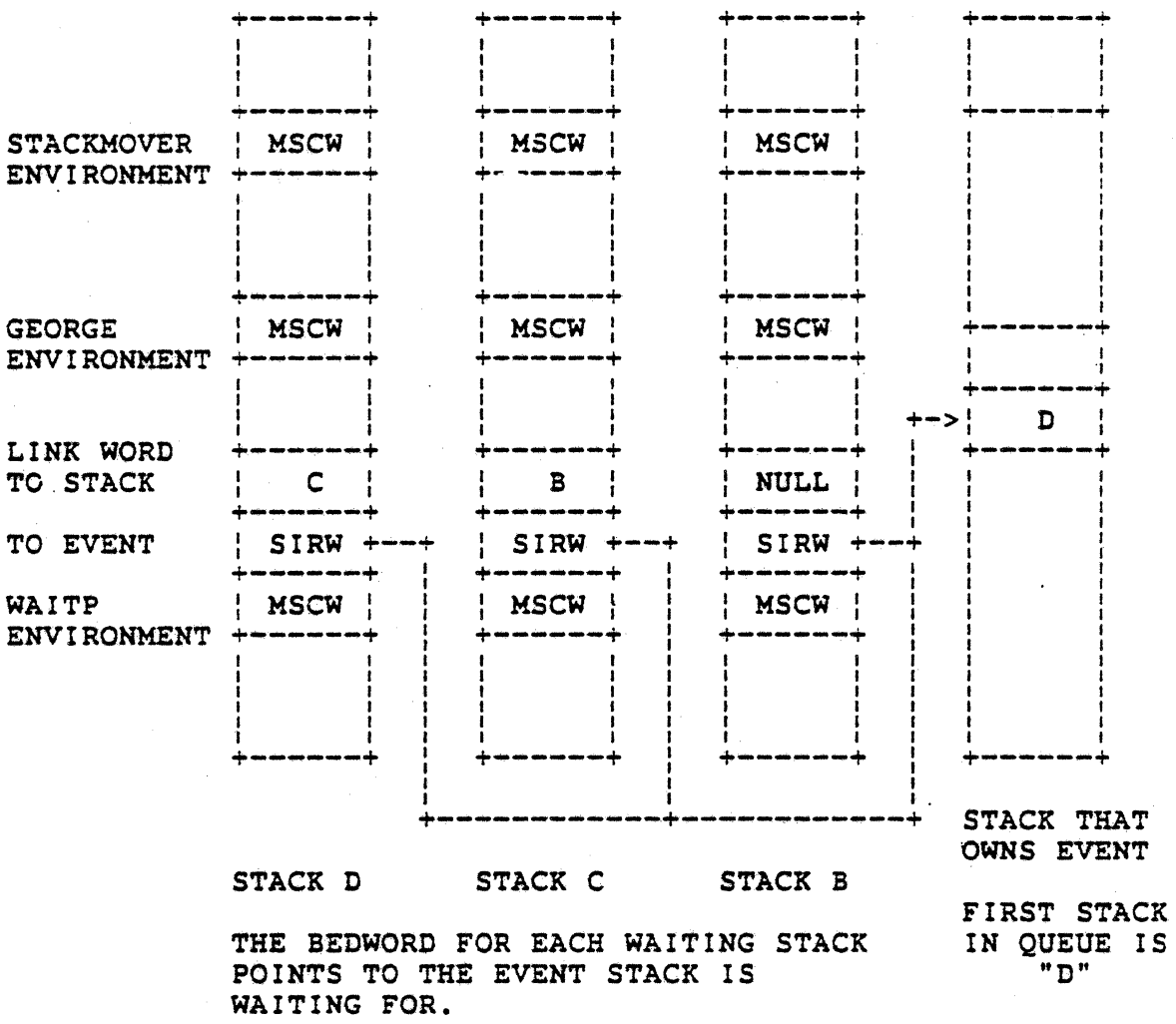


Figure 9-24. MULTIPLE STACK WAITING ON SAME EVENT

DELIVERY

When a user program wants to initiate a task the MCP procedure DELIVERY is called. The user program will pass to DELIVERY, two declared parameters followed by the parameter to be passed on to the called procedure and these parameters are followed by two undeclared parameters.

Shown below is a process statement extracted from the program PROCESS-CONTROL (figure 9-29) and the machine language code. Figure 9-25 shows the stack for the program shortly after the call on DELIVERY.

```
PROCESS ROOTER (A, B) [T1];
  MKST
  NAMC          DELIVERY
  ZERO          TYP, 0 INDICATES A "PROCESS."
  NAMC          LIFE, ROOTER'S PCW.
  STFF
  VALC          A, THE FIRST PASSED PARAMETER.
  VALC          B, THE SECOND PASSED PARAMETER.
  NAMC          T1, THE FIRST UNDECLARED PARAMETER.
  STFF
  NAMC          (2,0), THE SECOND UNDECLARED
                PARAMETER.  THIS IS THE CRITICAL
                BLOCK.
  STFF
  ENTR          (DELIVERY)
```

When DELIVERY was called, the parameter TYP was passed with a value of 0 for process. If this had been a call then the TYP parameter would have been a 1 and if this had been a run then this parameter would have been a 2. The next parameter passed was an SIRW called LIFE. This parameter points to the PCW to be entered. After LIFE, the procedure's parameters were passed, and in this case, they were A and B. After the procedure's parameters were passed, an SIRW was passed. This undeclared parameter points to the task descriptor for the procedure to be entered. Last, an SIRW was passed that points to the MSCW of the block that DELIVERY was called in. All this having been done, DELIVERY was entered and then INITIATEUSERTASK was immediately called. DELIVERY, itself, does nothing more. DELIVERY's real purpose is to accept a variable number of parameters that INITIATEUSERTASK will reach down and get. Notice that DELIVERY does not make any declarations of its own.

Figure 9-26. shows how the MCP continues with the process request. The task (PIB) was declared in the outer block of PROCESSCONTROL and partially set up by INITIATEUSERTASK. As INITIATEUSERTASK continues, DOCTOR is called and then DOCTOR

calls INITIATE which build the process stack for "ROOTER" and enters the stack number in the READYQ or SHEETQ, depending on various resources.

A more detailed MCP procedure flow is show in figure 9-27. Notice that if ROOTER had been called rather than processed the process stack of PROCESSCONTROL would have been topped off with GEORGE and STACKMOVER but as it is, INITIATE will exit back into DOCTOR which will exit back into INITIATEUSERTASK. INITIATEUSERTASK will exit back into DELIVERY and then back into PROCESSCONTROL.

PROCESSCONTROL, A PROGRAM

The program PROCESSCONTROL, shown in figure 9-28, and its associated external procedure RUNNER, shown in figure 9-29, were included in this manual simply to provide an example program that uses intrinsics and calls etc. This program's operation will be discussed briefly.

PROCESSCONTROL has declared three task arrays, four events, several variables and four procedures. This program demonstrates every way a procedure can be called. When PROCESSCONTROL is executed it will set variables A and B to 3 and 4, resp., and then take a programdump. This dump is valuable because, within the segment dictionary stack, the unused intrinsics descriptors can be seen. After the programdump, procedure ROOTER is processed passing A and B and T1. ROOTER starts by waiting for event EV0 to be caused so we don't have to be concerned with this procedure at this time. Since ROOTER was processed, PROCESSCONTROL continues by calling SINER passing task array T2. When SINER starts executing, event EV0 will be caused and SINER will then wait on event EV1. Now, ROOTER starts executing again and sets ROOT to the square root of A squared plus B squared. Next, ROOTER causes event EV1 and waits on event EV3. SINER has now started up again and SINE is set to the SIN of ROOT, a programdump is taken and event EV3 is caused. The dump just taken is useful because the intrinsics descriptors have been used and a comparison can be made between an unused intrinsic desc. and one that has been used. Since SINER has caused event EV3, ROOTER goes to the end of task. PROCESSCONTROL continues now that SINER is completely finished (SINER was called) and now COSINER is invoked (for lack of a better word). COSINER executes by setting COSINE to the COS of ROOT plus SINE and then exits. Now, ANSWER is set to ROOT plus SINE plus COSINE, another dump is taken and then RUNNER is run passing task array T3.

The preceeding paragraph emphasized how PROCESSCONTROL executed but did not go into the differences in the types of procedural calls. A general idea of these differences may be obtained from the table below:

	RUN	PROCESSED	CALLED	INVOKED
HAS OWN STACK	YES	YES	YES	NO
CALLING PROCEDURE CONTINUES	YES	YES	NO	NO
CALLED PROCEDURE IS COMPLETELY INDEPENDENT	YES	NO	NO	NO

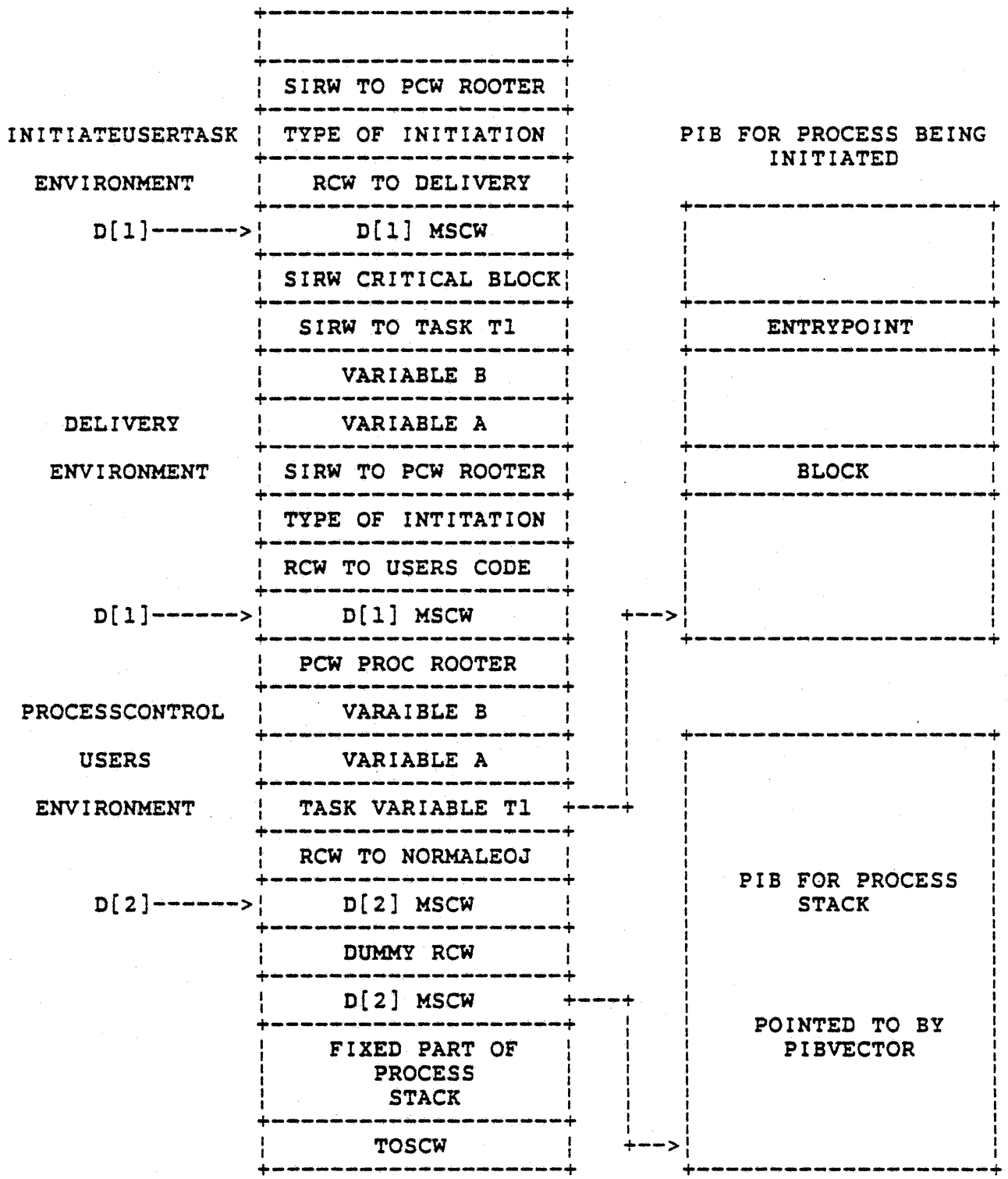


Figure 9-25. DELIVERY/INITIATEUSERTASK Procedures.

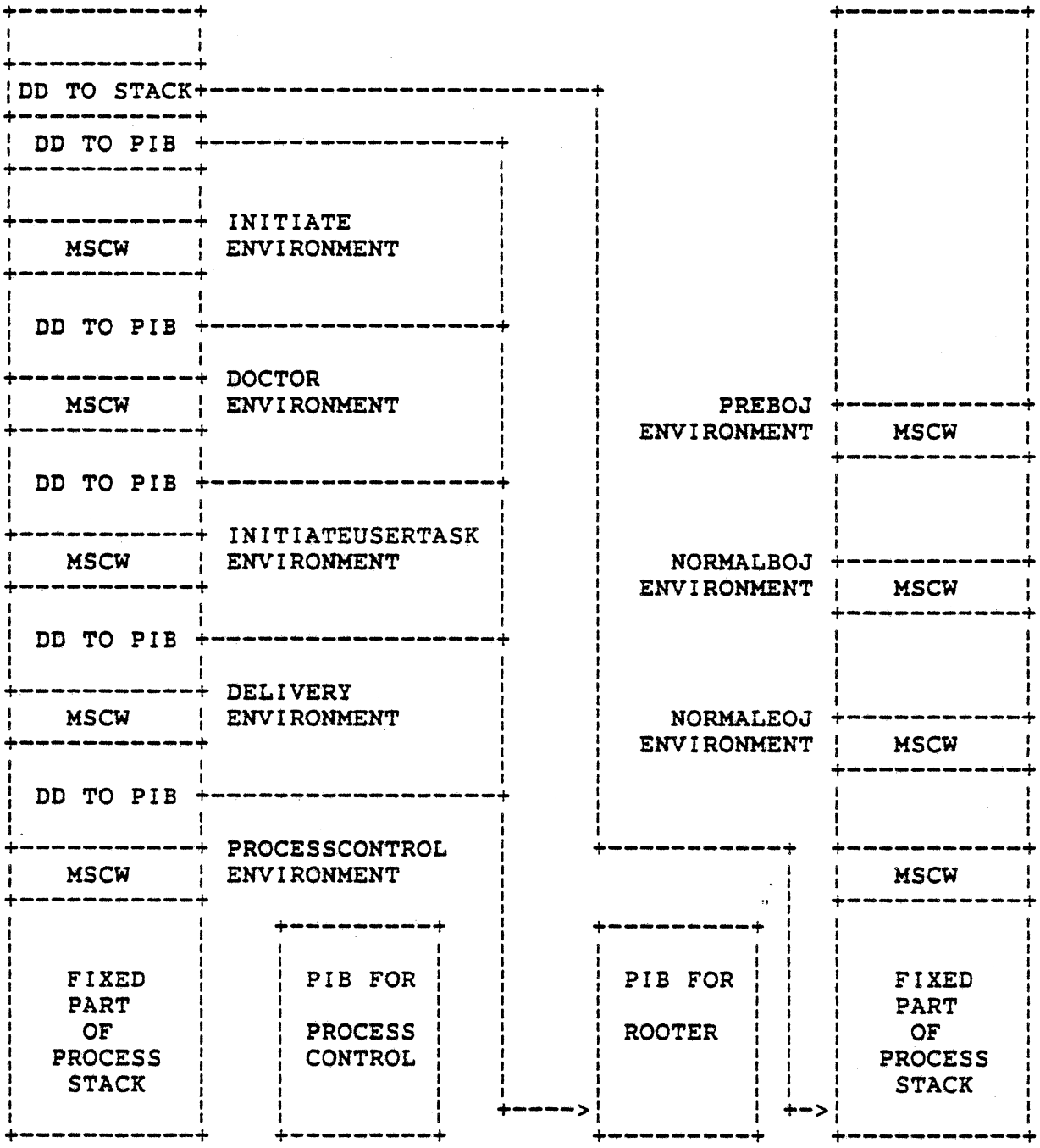


Figure 9-26. PROCESS STACK CONSTRUCTION

```
. THE PROGRAM
..DELIVERY
...INITIATEUSERTASK
....PARAMETERSTOTASK
....DOCTOR
.....INITIATE
......PICKASTACK
......GETSPACE
......RESURRECT
.......INSERTSTACK
.....GEORGE (IF "CALL")
```

Figure 9-27. PROCEDURE FLOW FOR USER TASK INITIATION

```
BEGIN
    TASK T1, T2, T3;
    EVENT EV0, EV1, EV2, EV3;
    REAL A, B, ROOT, SINE, COSINE, ANSWER;

    PROCEDURE ROOTER (A, B);
    VALUE A, B;
    REAL A, B;
    BEGIN
      WAIT (EV0);
      ROOT := SQRT (ROOT := A*2 + B*2);
      CAUSE (EV1);
      WAIT (EV3);
    END;

    PROCEDURE SINER;
    BEGIN
      CAUSE (EV0);
      WAIT (EV1);
      SINE := SINE (ROOT);
      PROGRAMDUMP(CODE, BASE, ARRAYS);
      CAUSE (EV3);
    END;

    PROCEDURE COSINER;
        BEGIN
          COSINE := COS (ROOT * SINE);
        END;

    PROCEDURE RUNNER;
        EXTERNAL;

    A := 3;
    B := 4;
    PROGRAMDUMP(CODE, BASE, ARRAYS);
    PROCESS ROOTER (A, B) [T1];
    CALL SINER [T2];
    COSINER;
    ANSWER := ROOT + SINE + COSINE;
    PROGRAMDUMP(CODE,BASE,ARRAYS);
    RUN RUNNER [T3];
END.
```

Figure 9-28. PROCESSCONTROL, the Program

```
PROCEDURE RUNNER;
  BEGIN
  REAL A, B , C;
  A := 1;
  B := 2;
  C := A + B;
  END.
```

Figure 9-29. RUNNER, the External Procedure

OUTLINE OF INITIATEUSERTASK

1. Get memory area for PIB if it has not been allocated.

2. Check to see if PIB is already in use.

3. If the process type is a run get a new PIB area. The old PIB is copied into the new PIB.

4. Move critical block MSCW (BLOCK) and SIRW to entry point PCW (ENTRYPOINT) to PIB.

5. Compute number of parameters.

6. Call DOCTOR passing the PIB as a parameter.

OUTLINE OF DOCTOR

1. Chain down SIRW's starting at ENTRYPOINT until a target is found. The target can be a PCW, operand or data descriptor. If a PCW is found a boolean called EASY is set. This is an internal procedure (internal to this program) being fired up. If an operand is found, the boolean EASY is reset. This is an external procedure being fired up. The operand's value is the segment dictionary index assigned to the procedure. If a data descriptor is found it is checked to see if it is a library descriptor. If it is, LINKLIBRARY is called to fire up the library. The search will continue for an operand or a PCW. At the end of the loop it is known if the process is internal or external.

2. HOSTNAME attribute is checked to determine if the program is to run on run on a different host.

3. The ORGHOSTNAME attribute is set.

4. A check is made to be sure the user is not doing a run on an internal procedure. If this condition is detected the initiator is DS-ED for non-external run.

5. Check to make sure a stack is not assigned to the PIB. If a stack is using the PIB, the initiator is DS-ED for "INITIATE ACTIVE TASK".

6. Some words in the PIB are cleared.

7. The job attribute TASKLIMIT is checked.

8. Various attributes are to be set. Before an attribute is inherited its validity bit is checked. If the validity bit is set it indicates that attribute has been set by the user and should not be inherited.

9. The SWAPSPEX word in the PIB is inherited. SWAPSPEX is the SUBSPACES attribute.

10. If the boolean EASY is set attributes are transferred from the parent. The attributes are USERCODE, NAME, COMPILERINFO (level of code file), STACKSIZE, file parameter block (FPB), PRIORITY, OPTION, MAXPROCESSTIME, MAXIOTIME, PRINTLIMIT, PUNCHLIMIT, MAXWAIT, and DISKLIMIT.

11. If EASY is not set, a procedure EXTERNALREFERENCE is called.

12. EXTERNALREFERENCE will do the following.

    a. Find the code file to be started.

    b. Read in the Segment Zero of the code file.

    c. Be sure the code file is code and executable.

    d. Check any parameters.

    e. If there is a PPB (Program Parameter Block) or FPB it is read into memory. The attributes in a PPB are assigned to the PIB by MUTATE. The FPB found is combined with any other FPB by COMBINEFPBS.

    f. Any attributes in Segment Zero are assigned to the PIB.

    g. If this is a compiler being initiated, the compilee's name is set.

13. If the type of initiation is RUN, the words for EXCEPTIONTASK, PARTNER, and BLOCK are cleared.

14. If the type of initiation is CALL and no partner is assigned, it is assigned to the initiator.

15. Some additional attributes are assigned.

16. The OPENCOUNT on the code file is incremented.

17. INITIATE is called passing the PIB.

18. The remaining code in DOCTOR is for an INITIATION of type CALL. This is where the parent will wait for the first CONTINUE.

OUTLINE OF INITIATE

A task can go thru INITIATE more than once. The first time it

is called from DOCTOR. If resources are not available the task will be scheduled. Later, ETERNALIR will call INITIATE again for the task.

1. If this is the first pass thru INITIATE some set up must be done.

   a. Call PICKASTACK to assign a mix number and stack number.

   b. If the task does not have a job file and needs one then call MAKEJOBFILE.

   c. Initialize the STACKINFO word.

   d. Put schedule priority in STACKSTATUS array.

   e. Put PIB descriptor in PIBVECTOR.

   f. Put this stack in PROCESSFAMILYLINK words.

   g. If this PIB has a critical block, add one to PROCESSCOUNTF of Software Control Word (SCW) of critical block.

   h. Parameters in the calling sequence are moved from the stack to a memory area (GETAREA).

   i. Compute size of process (D2) stack.

   j. If a CM or HS is outstanding, the stack will be scheduled.

2. Call GETSPACE for process stack memory.

3. If no memory is available, the task will be scheduled.

4. Move parameters to stack.

5. Build an IOCB in PIB for use by MCP.

6. Put PCW to GEORGE (PALACE) in PIB.

7. Put RCW's and MSCW's into stack so it looks like the stack has been used. The stack will contain an RCW to NORMALEOJ, an RCW to NORMALBOJ, and RCW to PREBOJ and GEORGE's environment.

8. Place the stack in the READYQ. From this point on the stack runs on its own.

OUTLINE OF PREJOB
-------- -- ------

PREJOB is an entry point in GEORGE. It will unlock the READYQ, clock the stack on the procesor and exit. It will exit into NORMALBOJ.

OUTLINE OF NORMALBOJ
------- -- ---------

NORMALBOJ runs in an environment with the D1 register pointing to the PIB.

1. Set up an overlay file for the stack.

2. See if there is a segment dictionary set up for this code file. The segment dictionary must be in the same box as the stack to be able to use it. If there is a segment dictionary set up, this stck will add one to the RUNNINGCOUNT and link to it.

3. If there is not a segment dictionary this stack can use, it must be set up.

   a. Call PICKASTACK to get a stack number.

   b. Initialize the STACKINFO entry.

   c. Set up a PIB.

   d. Call GETSPACE to get memory for the segment dictionary.

   e. Read the segment dictionary into memory.

   f. Fix any MCP intrinsic references. They are changed to SIRW's into the MCP stack.

   g. Set the RUNNINGCOUNT to one.

4. Write BOT log record.

5. Take the entry point PCW, change it to an RCW and exit into the user's program.

OUTLINE OF NORMALEOJ
------- -- ---------

1. If the state of the stack is that it was initialized by the library linker and this is still the state then it never froze. That is, a stack invoked the library and it started. However, the stack never did a FREEZE, thus the invoker never got linked and will not be linked to the library. All stacks waiting on the FREEZE are DS-ED.

2. If the task was DS-ED and had a stack overflow or

exceeded memory, the proper message will be generated. These messages are displayed now because the stack has been cut back and there should be enough stack and memory.

3. If the task was a compiler and the task value is non-zero, a syntax error occured. The HISTORY word is updated to indicate normal EOT or syntax.

4. If the stack is capable of being linked to a DBS, records owned by this stack must be released. A search is made for all data base stacks running on the system. Each DBS is called to free the records.

5. If a TASKFILE is present, it is closed. The area for the FIB is released.

6. If any disk file headers are still locked by this stack they are liberated.

7. If the stack still has units attached, they must be released. A search is made thru the UNIT table for the units we own. These units are released.

8. If the task was a visable task, it logs EOT. That is, an EOT record is written in the LOG.

9. If the stack still has any job messages they are released. A job message is an Accept or Display line.

10. The memory estimate is updated.

11. If the stack is linked to the intrinsics stack, the RUNNINGCOUNT of the intrinsics is decremented. If this is the last stack using the intrinsics, it is terminated. TERMINATEDISTACK is called for this function.

12. The RUNNINGCOUNT of the segment dictionary is decremented. If it is zero, the segment dictionary is terminated.

13. The overlay file is released.

14. The owner of the stack is changed to the MCP.

15. FORGETCHECK is called to make sure there are no in-use areas of memory left around. If there are, they are made save areas. A dump is taken.

16. Mix numbers are cleared from the PIB.

17. If the stack was visable, a message is sent to CONTROLLER. The stack is going to EOJ. In the case of job stack, the CONTROLLER will start the printing.

18. If the stack is a SWAPJOB, SWAPPER is told to take the stack and GEORGE is called so the processor can get off of the stack.

19. For other stacks, a message is sent to ETERNALIR telling it to take this stack. GEORGE is called to process switch to another stack. Thus, all that is left of the stack is a stack and a PIB. A message is in the queue for ETERNALIR. The processor has switched out of the stack.

20. ETERNALIR will get the message to terminate the stack. It will call TERMINATE.

OUTLINE OF TERMINATE
------- -- ---------

1. If the PIB was generated by the system, it will be released by TERMINATE.

2. If the PIB was declared by a user it will be cleaned of some of the information.

3. If there is a critical block for this stack, the PROCESSCOUNT in the tag 6 word above the critical block MSCW will be decremented by one.

4. The stack is removed from any PROCESSFAMILYLINK.

5. The EXCEPTIONEVENT is caused for this stack.

6. If the type of process was a CALL, a CONTINUE is done to the CO-ROUTINE.

7. If it was not a SWAPJOB, the memory area for the stack is released.

8. An event, STACKFINISHED, is caused.

OUTLINE OF GEORGE
------- -- ------

GEORGE is a define that calls a procedure called GEORGE declared in SOPHIA. While running in GEORGE, the D1 register points to the PIB.

1. The PROCESSTIME the task has picked up in this slice is computed. This is added to the total PROCESSTIME.

2. The fine priority is adjusted for the task. This priority will bias the processor for I/O bound tasks. That is, an I/O bound job will have a higher fine priority than a CPU bound job.

3. If the parameter (MANDATE or EDICT) state something special is to be done, it will be handled.

   a. Get off the processor (the stack may want to go to EOJ) in which case GEORGE will branch down and look for a stack to move to.

   b. The parameter may say just log off the processor and return. This is done in NORMALEOJ to get a final processor time for the stack.

   c. The other special case involved building a PCW so new stacks can enter GEORGE at a certain point. This is done in initialization.

4. A bit is set to indicate the stack is in GEORGE.

5. If this is a swap job and its elapsed time or process time are up it will be swapped.

6. If the parameter said to swap the stack now, it will be swapped.

7. If the stack has a PROCESSTIME limit, it will be checked. If it is too large, the stack will be DS-ED.

8. If the stack has an IOTIME limit, it will be checked. If it is too large, the stack will be DS-ED.

9. If it has been greater than one second since ONESECONDBURDERN has been called, it is called.

10. If there are messages linked from the COMQHEAD for this CPM they are done at this time. On a B7000 this is for finishing off I/O operations. The basic set up is as follows. All IOCB's are allocated in Global memory. This allows any CPM to look at all IOCB's. In fact, CPM's will pick up I/O result queues for IOM's they do not own. The CPM that picks up a result queue will try to finish off all IOCB's in the queue. If it finds one with a non-zero box number in IOCB[IOCBCONTROL] and that number is not the same as the processor number doing this, it will be queued for the proper CPM. The IOCBCONTROL word is set as follows. If a task and its buffer are in Global, the IOCBCONTROL word is set to zero. Any CPM can handle it. If the buffer is in local, the task must be in local and the IOCBCONTROL word is set to the box it is running in. In the case of a local buffer a check is made to be sure that the box has a direct path to the unit.

11. If the READYQHEAD is non-zero there is a task in it that wants to run. We will look at the READYQ and decide what to do.

12. If the stack state of this stack is ALIVE and the

priority of the stack the processor is in is better than the priority of the top entry in the queue, the stack the processor is in will maintain control of the processor. Otherwise, the stack the processor is in will be placed in the READYQ and we will find a stack to move to. If this stack holds the processor, it branches down in GEORGE.

13. To find a stack to move to the processor looks at stacks in the READYQ. It must find one that the processor can move to. That is, one in the same box as the processor or in Global. If one is not found, the processor branches to the idle loop. If one is found, we rebuild the linkage to link around the stack to be run.

14. A STACKMOVER is done to the stack that was found. This is an entry and exit point for GEORGE. That is, this call on STACKMOVER, which does a MOVESTACK and an exit, is where the processor will enter when the stack gets a processor back. GEORGE branches to the exit point.

15. If there are not active stacks, the idle loop starts.

16. If there is an idler stack for the processor, it is moved to.

17. The interval timer is set.

18. The EGG timer is reset.

19. The idle pattern is displayed and the processor idles.

20. If the READYQ is not zero, we go look at it.

21. If this stack is alive, we get ready to exit back to it.

22. If there is nothing to do, GEORGE branches up to do a ONESECONDBURDEN.

23. The exit code for GEORGE follows.

24. The stack is clocked on the processor.

25. The interval timer is set.

26. The ACTIONQ for the stack is checked. The ACTIONQ may tell GEORGE to delink events, call KANGAROO or process software interrupts.

27. Exit to user.

OUTLINE OF ONESECONDBURDEN
------- -- ---------------

1. Make sure all CPM's are still running. If one has not checked in take it off-line.

2. Update utilization statistics.

3. If the time of day is greater than one day, change the date.

4. Wake up stack waiting on time.

SECTION 10

INPUT/OUTPUT OPERATIONS

INTRODUCTION

This section is divided into 3 sub-sections. The first sub-section is an overview of the I/O subsystem, the second is concerned with a program named FILEEXAMPLE and the last is a detailed account of I/O procedures flow involved in reading a card.

I/O SUBSYSTEM OVERVIEW

There are two main characteristics of the logical I/O design which are responsible for its speed and obscurity. First, the memory area containing the FIB is treated as a stack rather than an array. Stacks are by far the more efficient types of addressing since they require no explicit indexing while arrays do. Put another way, while I/O routines are running they assume that display register 1 will be pointing at the base of the FIB. Thus, they can address FIB variables directly as stack locations (1,X) rather than TABLE [X] where "TABLE" is an array. This means that NEWP can do VALC 1,X instead of NAMC FIB,LT8 X, NXLV which is clearly more efficient.

The second characteristic of the I/O routines involves decision making. Since file handling is a complex business with many possibilities for variation, a large amount of decision making is necessary. Some of these decisions are: read or write? forward or reverse? blocked or unblocked? fixed length or variable length? translate or no translate? word or character oriented? direct I/O or not? sequential or random?

Lots of questions mean lots of code which costs time and leads to large, unmaintainable and unfathomable routines. The approach taken in our MCP is to extract the decision making out of the highly used nitty-gritty record by record routines and make as many decisions as possible when the file is opened or when the file undergoes a change of state. This sounds

like a very simple and obvious idea, but it requires some pretty exotic (though still basically simple) code. It also makes the I/O routines less readable to a newcomer, which may explain why it is not widely understood.

NO DECISION MAKING ASPECT

Intuitively, one suspects that a lot of decisions are made at file open time; however, there are many details which must be handled at the time of the actual I/O statement. Almost everyone appreciates that there are multiple buffers and they must be rotated, and the file may be blocked. Also there are further details which may be dependent on the device or file organization.

The real problem in handling these details is when they change state. For example, if a file was being read and now a write occurs, or the file was being accessed randomly and suddenly there is no key. There are hundreds of circumstances which may cause one of these changes and unfortunately it is very difficult to put this special case code anywhere but at the point where the special case occurs. Compilers cannot detect it and the operating system cannot predict it. If the programmer switches from reads to writes, it must be handled by the write procedure. In other implementations this requires the write procedure to grow very large (and slow and hard-to-maintain) as the number of special cases grows. The same is true of read, seek, space, etc.

Even if no special cases occur, it is easy to see why the routines can get large. On a request for any given record, a decision must be made on whether the record is in the buffer or whether an actual I/O must be done. A decision must be made on whether a buffer is available due to the use of this record i.e., last record in a block. A decision must be made on where to move this data record, and whether it is in the correct character set (INTMODE). A decision is made on moving a record pointer forward or backward, depending on whether the records are word aligned. A decision is made on what to do if there is a parity error, etc. Is it variable-length record? Did the programmer make a mistake? For example, perhaps he asked for 10 words and there are only 10 characters. There are hundreds of these details to be worked out when I/O is being done and the job of the I/O handler is to make this work be done as efficiently as possible.

This lengthy set of questions is intended to emphasize the decision making that occurs in I/O operations. It is just such decision making that the FIBSTACK implementation seeks to avoid. There are two aspects of this implementation. The first aspect involves the decision making which happens when the file changes state (from read to write for example). The strategy is to consider the various requests that can be made of I/O as corresponding to states of the file. Thus a file

that was being read serially would be in the read-serial state. In order to do a random write, the file would have to undergo a change in state, to the write-random state. The routines function as if they are operating on the following table (actually non- existent):

REQUEST	WRITE SEQ	READ SEQ	WRITE RANDOM	READ RANDOM
CURRENT STATE				
WRITE SEQUENTIAL	DIAG.	COLUMN	COLUMN	COLUMN
READ SEQUENTIAL	COLUMN	DIAG.	COLUMN	COLUMN
WRITE RANDOM	COLUMN	COLUMN	DIAG.	COLUMN
READ RANDOM	COLUMN	COLUMN	COLUMN	DIAG.

As long as the request being made corresponds to the current state of the file, no special case code is required and the file does not change state. This is the situation that occurs along the DIAGONAL of the table. If user I/O receives a request that is different from the previous request, i.e., requires a different file state then that request is off the diagonal and requires special handling.

It is easy for compilers to determine the type of request that needs to be made. Therefore the compiler can choose the correct routine of several user I/O types. For example, if the compiler scans READ (F, ...) it generates a call on the read-serial routine and if it scans WRITE (F[X],...) then it generates a call on the write-random routine.

When we speak of compilers making calls on routines, we know that what is actually meant is the compiler generates a name call on the location that contains a program control word (PCW). In our implementation of user I/O, all the compilers know is a set of locations that corresponds to the set of possible requests for I/O. Thus if a compiler sees READ(F, ...) it generates a call on the appropriate location. The routine actually entered depends on what PCW is in the referenced location.

The key to FIBSTACK is that the MCP manipulates the PCW's contained in these locations so that the compilers need to know only which location to call and user I/O ensures that the correct PCW is there. In particular, it ensures that as long as the requests fall on the DIAGONAL, the referenced location will contain a PCW pointing to a simple routine with no special case code. If a request is made off the DIAGONAL, then a change in state is being requested and the PCW in that

location points to a larger routine that contains the special case code necessary to perform the state change and do the requested I/O.

In reality, the compilers make six different requests of user I/O. The types are:

READ SERIAL

WRITE SERIAL

READ RANDOM

WRITE RANDOM

READ REVERSE

SEEK

If the file is in a NOT OPEN state, then any request will be off the diagonal and will require special handling. For all other states, if the request stays the same then the handling is simple. Now if we consider the columns of the table to be equivalent to the locations accessed by the compiler we can see how FIBSTACK manipulates the PCW's in order to minimize the decision making. It is only necessary to ensure that when the next request is received from the program, it will cause entry into a simple routine if the request is on the diagonal or into a more complicated one if it is not. So far any given file state, the locations accessible by the compiler will look like a row of the table. Each row will contain a pointer (PCW) to an easy routine and the other PCW's will point to routines that change state. Also, when the file changes state, the locations contain a different row of the table.

If we consider a file to be in a write-random state the row would look like

COLUMN

COLUMN

DIAGONAL

COLUMN

The easy PCW is the diagonal in the write-random column and the write-random row. The non-diagonal routines are called COLUMN routines. Thus, there is a write-serial column routine that will be entered when a write serial request is received and the file is not in the write-serial state. Also, a read-random column routine and a seek column routine, etc. The read reverse and seek PCW's are not shown above or in the table because they are always COLUMN procedures. The diagonal routines are the real workhorses and are very small and to the

point. There is as little special case code as possible in diagonal procedures.

The second aspect of the no decision making implementation arises out of the desire to make fewer decisions on each individual record. These are the decisions about blocking, record lengths, variable length records, translations, etc. Which are often made to choose one line of code rather than another, or one path rather than another. These decisions individually do not seem very important or expensive and yet the routines seem to grow larger and larger, and slower and slower. The solution taken in the FIBSTACK implementation is to make separate routines in many cases where a decision would have seemed easier. Thus there are a whole set of routines that include software translation, another whole set that does not. Neither set needs to make a test to see if translation is required, and neither set carries the extra code necessary to do its partners job. There is a whole set of routines which deal only with file type three (a type of variable length structure) and another set for file type two (another variable length strategy), and a set for file type zero (fixed length records), etc. There are different routines for blocked and unblocked files, different routines for character and word oriented files, and routines that release the buffers and those that do not.

In short, a great part of the decision making has been moved. Rather than making them at record access time, we make them at file open (or file set up) time.

It is important that diagonal routines be as compact and efficient as possible. There is a read-serial routine, a write-random routine, etc. There are column routines to handle switching off the diagonal. It should be noted that there are hundreds of diagonal routines (i.e., lots of read-serial diagonals, lots of read-randoms, etc.). File open makes the decisions about the file necessary to select the appropriate set of diagonals (blocked/unblocked, translate/no translate). From then on, the MCP will arm one of the possible request with the appropriate diagonal PCW and all other request with PCW's to column routines. If the diagonal PCW is entered, the routine entered would be for example

SERIAL READ-FILE TYPE 0-WORD ORIENTED-BLOCKED-NO TRANSLATE-RELEASE

Obviously, a lot of decisions have been made already when this routine is entered.

All of these diagonals are little procedures local to procedure FIBSTACK. They are declared in a very precise order. There is always a write routine followed by a read routine to make up a pair. There is always a word oriented pair followed by a character oriented pair, and there is one of each of these for blocked and for unblocked files:

WRITE UNBLOCKED WORD

READ UNBLOCKED WORD

WRITE UNBLOCKED CHARACTER

READ UNBLOCKED CHARACTER

WRITE BLOCKED WORD

READ BLOCKED WORD

WRITE BLOCKED CHARACTER

READ BLOCKED CHARACTER

There is one of these little binary ordered sets of procedure declarations for each of the possible file types (there are 8). This makes 64 diagonals. Then this pattern is repeated with soft translation. That, theoretically makes 128. These would be repeated without buffer release, making 256. All of these procedures are declared in FIBSTACK as local procedures. At initialization time, FIBSTACK is called and sets up the global array IOPCWS in which it puts copies of all its PCW's (see figure 10-1). Now, considering the binary relationship in the ordering of the PCW's, file open can compute an index into IOPCWS which will locate the PCW's needed. The index is computed as follows:

0 & CHARECORD [1:1] & BLCKED [2:1] & FILETYP [5:3]

It should be pointed out that there are not 256 diagonals. Some of them would fall into illegal slots while others are so weird that they are not reasonable. However, FIBSTACK and OPEN and SETUPFIB know about these discrepancies and allow for the missing diagonals. Another point is that these routines generally apply only to the serial read and serial write diagonals. For random read and random write, there are a small number of routines. When these random routines are entered they adjust the record location and then go to the corresponding serial routine which actually moves the record around.

The second characteristic of the FIBSTACK implementation is to use the FILE INFORMATION BLOCK (FIB) as a stack. This is not a difficult concept, but it has several facets that interact, making it hard to explain. One MCP feature that is utilized is the PSEUDO-STACK mechanism.

There are 16 pseudo-stacks. These 16 stacks are used by the MCP to address all of memory with stuffed indirect reference words (SIRW's). With these 16 stacks it is possible to address 16 different areas, each 4"10000" words long. The first pseudo- stack addresses memory from 0 to 4"0FFFF", the second pseudo-stack addresses memory from 4"10000" to

4"1FFFF", etc. Thus, any given word can be addressed as an offset in one of these stacks. M[4"17843"] = pseudo-stack 1, offset 4"7843". Similarly, any memory address can be split up into its component pseudo-stack number and offset by merely dialing out the high order hexade and adding the base value (PSEUDOSTACKBIAS) of the pseudo-stacks to get the actual stack number. Thus, 4"57962" = stack PSEUDOSTACKBIAS+5, offset 4"7962". This feature lends itself to the use of SIRW's to address memory. The only restriction is that one has to know the location of a MSCW in the stack since SIRW's address relative to an MSCW.

This is one way in which FIB's are used. When a FIB is built (at some arbitrary location, e.g., 4"12345"), that FIB contains an MSCW at FIB[1] and every word in the FIB may be addressed by an SIRW that locates that MSCW and gives an index relative to it. For our example, the FIB at M[4"12345"] would have an MSCW at M[4"12346"] (the second word in the FIB) and the stack number in the SIRW would be PSEUDOSTACKBIAS+1 and the displacement field in the SIRW would be 4"2346".

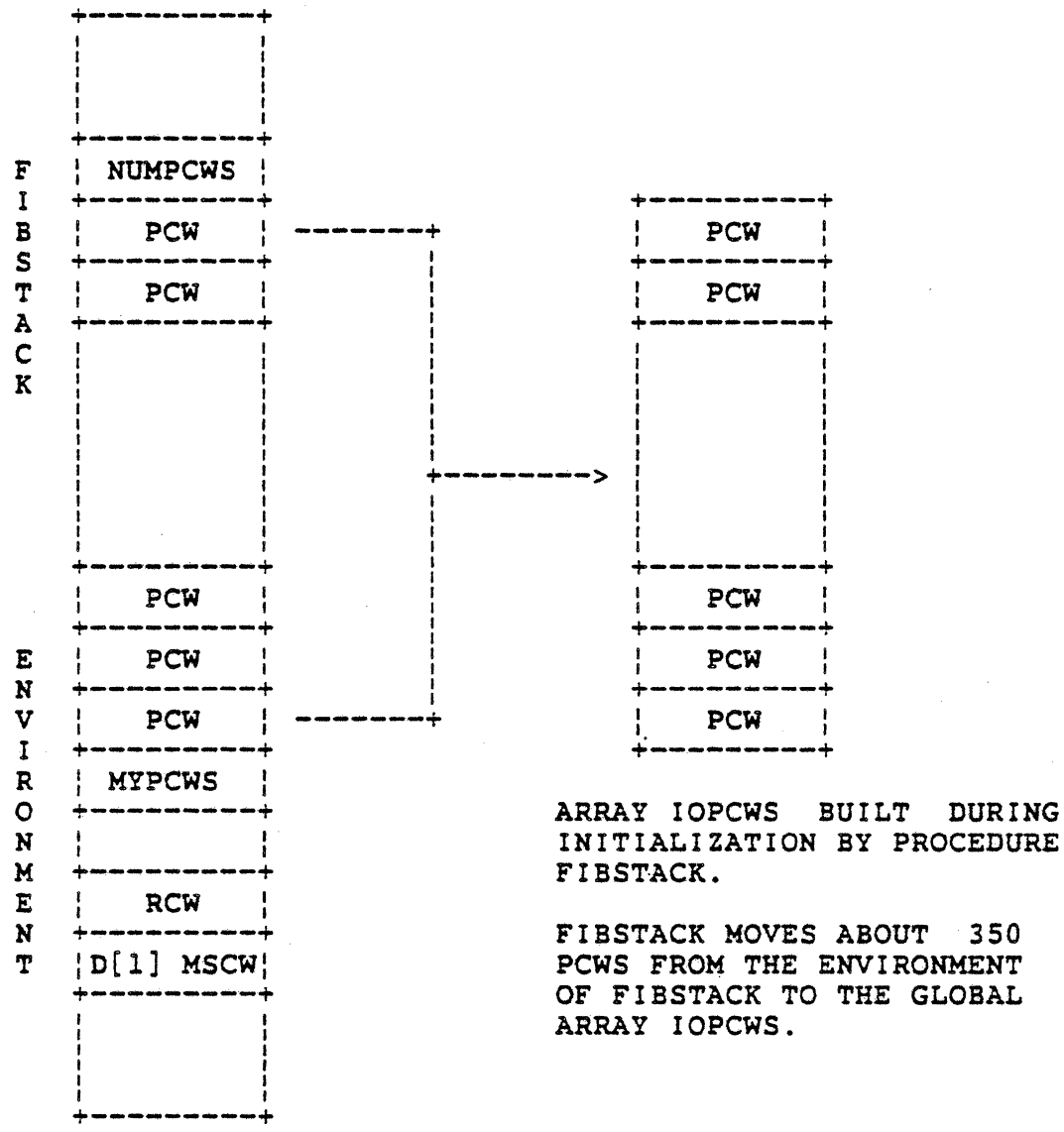


Figure 10-1. IOPCWS Array

Throughout the discussion of how decision making is avoided in MCP I/O functions, reference has been made to the locations where compilers expect to find PCW's pointing to routines that handle I/O. In actual fact, these PCW's are in the FIB for a given file. They are accessed by the compilers via an SIRW (see figures 10-2 and 10-3). This SIRW is located at FIB[0] and points to one of the FIB's PCW's. This SIRW is called SELECTOR because the compilers use it to select the request that has actually been made by the program. The selection is made by altering the low order 3 bits of SELECTOR. This picks one of the six possible requests (PCW's). Typical code for a read or write statement is:

```
MKST

ZERO

NAMC FIB

INDX

LOAD

LT8 <SEL>

LOR

<P1>

<P2>

<P3>

<P4>

ENTR
```

FIB, here, is the stack location generated for the file and contains a data descriptor pointing to the FIB. <SEL> is a literal indicating the request (0-serial write, 1-serial read, 2-random write, etc.). P1, P2, P3, and P4 are parameters passed to the MCP I/O routines. Notice that the load gets FIB[0] which is SELECTOR. SELECTOR points at the first PCW we may require. These PCW's have the pseudo-stack number of the FIB they are within, in their stack number field and their lexic level field contains a 2. When one of these PCW's is entered by the above code, the hardware will detect that the PCW points into a different stack and, as a result, will cause a display register to point to the same MSCW that the SIRW points to, i.e., the FIB MSCW. Since this is a level 1 MSCW, display register 1 will be caused to point at the MSCW in the FIB. The D2 register, of course, addresses the MSCW in the process stack (placed there by the MKS™ instruction in the code above).

Thus, when one of these procedures is entered, the hardware will perform the additional function of ensuring that the entered procedure will be able to address those quantities declared at the same lexic level as the procedure itself. This situation exists when any of the procedures which do I/O are entered via the PCW's in the FIB.

FILEEXAMPLE, A PROGRAM

The program, FILEEXAMPLE, shown in figure 10-4 will be used here to give a more direct example of how the MCP handles user I/O request. The program does little more than reading a card and writing it to disk but in doing so, it opens up many areas of discussion.

FILEEXAMPLE declares two files, MYCARDFILE and MYDISKFILE, and one array, CARDIMAGE. When the compiler compiles this program, the information in each file declaration will be placed, in an encoded form, into a data pool and written into the program's code file. When FILEEXAMPLE is executed, its stack building code will cause data descriptors that reference these data pools to be placed in the process stack. As far as a process stack is concerned, a file is simply a data descriptor and nothing else.

When each of the files is first referenced, the MCP will become aware that the file has not been set up in memory and will do so. When the file is referenced for the first time the MCP will read in its data pool and distribute the data pool information between two arrays, the FILE INFORMATION BLOCK (FIB) and the LABEL EQUATION BLOCK (LEB), and then replace the original data descriptor (the one pointing to the data pool) with a descriptor that points to the FIB. This can be seen in figure 10-2.

Other locations are also initialized in the FIB and LEB. The PCW's shown in figure 10-2 come from the IOPCWS array previously discussed. The PCW's currently in the PWRITES, PREADS, PWRITEN and PREADN FIB locations are ""COLUMN"" PCW's. The ""TANK"" shown in this figure is not set up at this time.

After the FIB and LEB are set up, an exit is made back to the user program and, in the case of a card read, the PREADS location is entered. PREADS, being a COLUMN procedure at this time, will be responsible for locating the file and sorting the physical unit number in IOINFO in the FIB. The TANK shown in figure 10-2 is now set up and FILESTATEF in the FILESTATUS word is set to 3 indicating the file is now in the serial read state. Once the buffers are filled for the first time, the first buffer is transferred to the CARDIMAGE array and another I/O initiated, PREADS is replaced with a specific DIAGONAL PCW and an exit made back to the user program (FILEEXAMPLE). On all subsequent card reads, the DIAGONAL procedure will be

entered. There is no further need to locate the file, set up the buffers, etc.

When the first write to disk is initiated, the same basic flow as for the previously discussed card read is performed. The first reference to the disk file descriptor will cause the MCP to read in the disk file's data pool and set up the FIB and LEB accordingly. When the FIB PWRITES location is first entered, a COLUMN PCW will be in that location and execution of this procedure will cause the disk TANK to be set up. Since the file was declared with two records per block (buffer), a physical write to disk will not be performed and, at this point in time the FIB DHEADER word will contain a 0, no header will be set up yet and no user disk obtained.

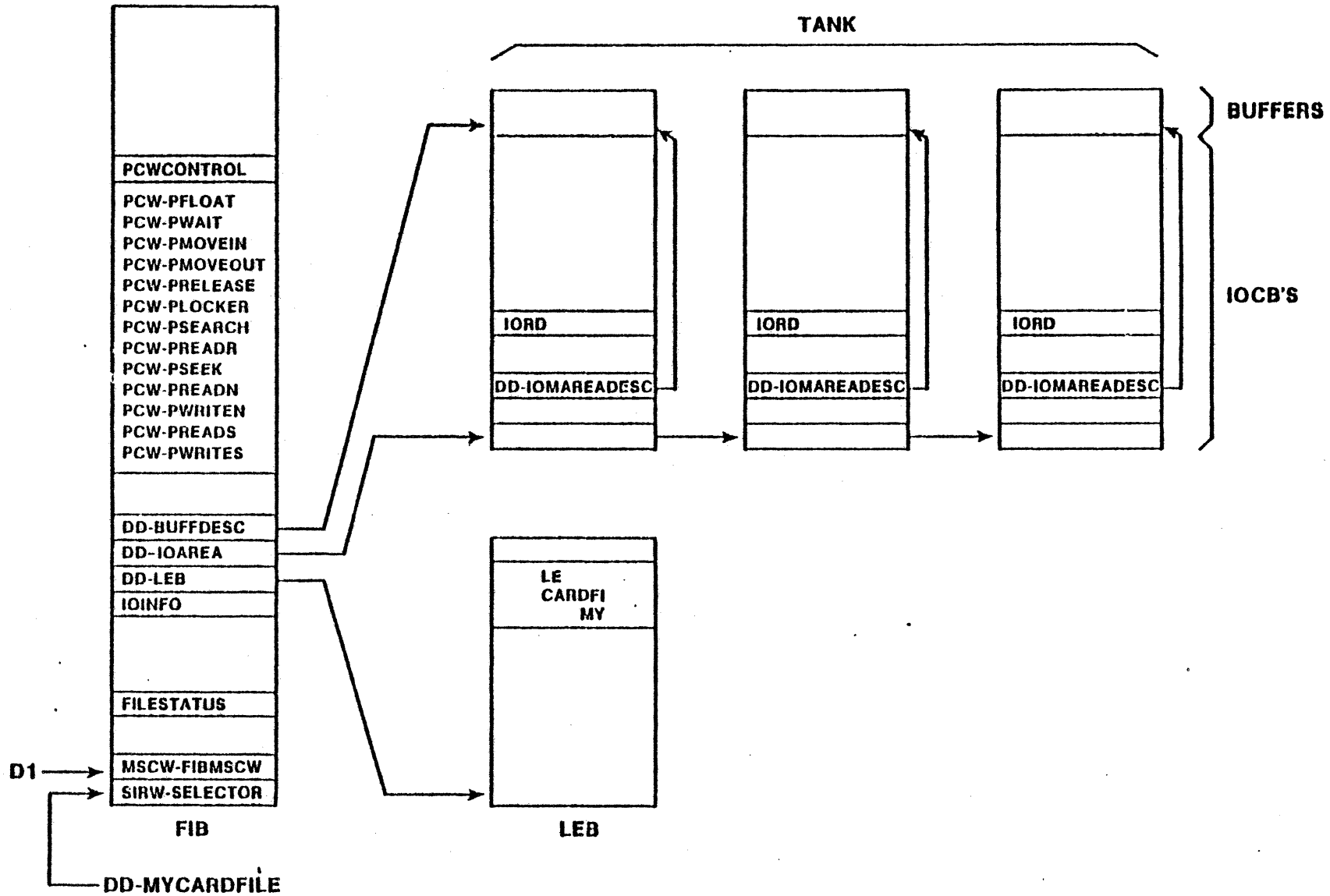


Figure 10-2. MYCARDFILE

Figure 10-2. MYCARDFILE

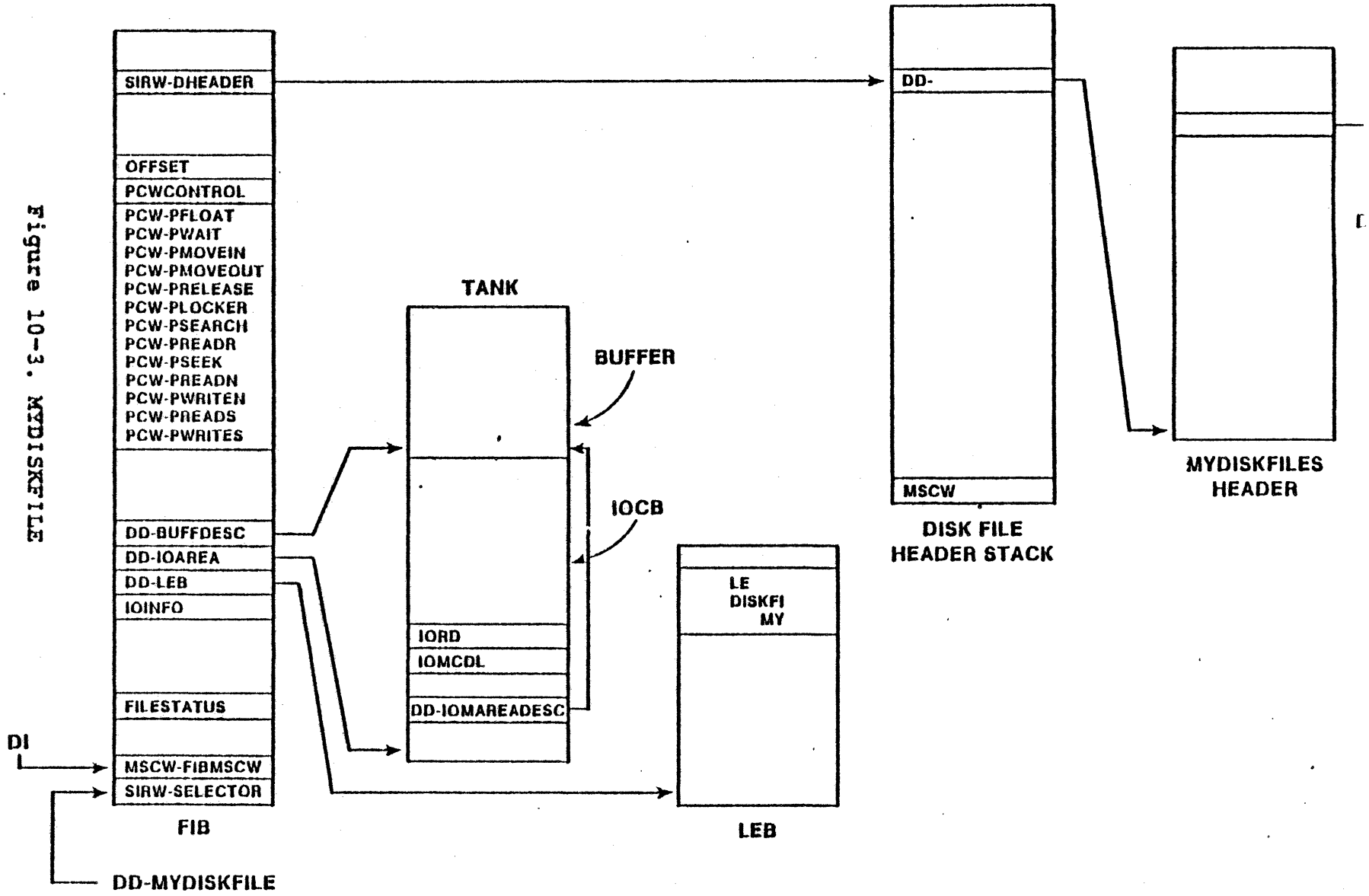


Figure 10-3. MYDISKFILE

Figure 10-3. MYDISKFILE

After the COLUMN procedure has transferred the CARDIMAGE array to the buffer, the PWRITES location is replaced with a DIAGONAL PCW and an exit made back to the program. The next time a record is written to this file, user disk will be obtained, a header set up and DHEADER will be replaced with an SIRW pointing to the appropriate DISK FILE HEADER STACK location.

As cards are read and written to disk, a card end-of-file will eventually come and the end-of-file branch taken to CARDEOF. At this time, the disk file is locked and the program ends. When the LOCK occurs, the header will be written into the FLAT DIRECTORY, an entry made in the ACCESS STRUCTURE and FORGETSPACE called for the HEADER and TANK. The FIB and LEB will remain in memory after the LOCK and will not be removed until the block is exited.

DATA POOL

Figure 10-5 shows the data pool created by the ALGOL compiler for MYCARDFILE. When the file is to be set up, an MCP procedure named ATTRIBUTEHANDLER will, via PRESENCEBIT, cause the data pool to be brought into memory and pointed to by a local data descriptor named LIST. The first word in the data pool contains miscellaneous information. This is followed by the file name and that, by an attribute list.

Attribute lists consist of a series of entries, all with the same basic format the first character is the length of the entry, the second is the attribute number and the last part of the entry is variant information associated with the attribute; for instance, variant information for the buffer attribute is the number of buffers.

FILE OPENING AND HARDWARE INTERRUPTS

The MCP is notified of a file opening in a very unique way a hardware interrupt. This is accomplished by trying to index a data descriptor that is already indexed.

Write statement code, for instance, is generated by a compiler with the assumption that the FIB is set up. If this is the case, then, when the FIB's descriptor is indexed by 0 with the intention of loading SELECTOR to the top of the stack, the index will work and everything goes as previously described. If this is not the case, however, an interrupt will occur.

Figure 10-6 shows the process stack for FILEEXAMPLE immediately after executing its stack building code. The data descriptor at 2,3 is for MYDISKFILE and the descriptor at 2,4 is for MYCARDFILE. The address and length fields of these

descriptors are perfectly normal but notice that the size field contains a seven. This is not valid. The index bit is also set. This in itself is not incorrect but an interrupt will occur if the hardware tries to index a descriptor that is already indexed.

```
$ SET LIST STACK CODE
BEGIN
    FILE MYCARDFILE (KIND       = READER,
                     BUFFERS    = 3,
                     MAXRECSIZE= 14);
    FILE MYDISKFILE (KIND       = DISK,
                     BUFFERS    = 1,
                     BLOCKSIZE = 60,      % WORDS
                     AREAS      = 1,
                     AREASIZE   = 90,     % RECORDS
                     MAXRECSIZE= 30);     % WORDS

    ARRAY CARDIMAGE[0:29];

    LABEL BACK, CARDEOF;

    PROGRAMDUMP(ARRAYS,FILES);

BACK:
    READ (MYCARDFILE,14,CARDIMAGE) [CARDEOF];
    WRITE (MYDISKFILE,14,CARDIMAGE);
    PROGRAMDUMP(ARRAYS,FILES);
    GO BACK;
CARDEOF:
    LOCK (MYDISKFILE);
    PROGRAMDUMP(ARRAYS,FILES);
END.
```

Figure 10-4. FILEEXAMPLE

```
                                    010000000000
                                    0E01010AD4E8  Standard
                                    C3C1D9C4C6C9  Form
                                    D3C500000000  Name
                                    031D04030809  Attribute
                                    031A03030F0E  List
                                    000000000000

 LIST[0] - 47:8 = LEVELF, THE LEVEL OF THE FILE DECLARATION.
           07:8 = SPRACHEF, THE LANGUAGE THE FILE WAS DECLARED IN.
                  0 = ALGOL.

LIST[1] - THE FILE'S INTERNAL NAME IN STANDARD FORM.  THE NAME IN THE
 THRU     EXAMPLE ABOVE IS "MYCARDFILE."
LIST[3]


LIST[4] - THIS IS THE START OF THE ATTRIBUTE LIST, SOMETIMES REFERRED
          TO AS THE FILE'S DESCRIPTION.  THIS LIST CONSISTS OF A
           VARIABLE NUMBER OF ENTRIES WITH THE FOLLOWING FORMAT:

          CHAR. 1 = NUMBER OF CHARACTERS IN THE ENTRY(SELF-INCLUSIVE)
          CHAR. 2 = ATTRIBUTE NUMBER.  USED TO INDEX THE
                    "ATTRIBUTETABLE."
          CHAR. 3 = VARIANT INFORMATION ASSOCIATED WITH THE ATTRIBUTE.
                    THE LENGTH OF THIS PORTION OF THE ENTRY IS (VALUE
                    IN CHAR. 1 MINUS TWO) CHARACTERS.

THE FOLLOWING IS A DECODE OF THE ATTRIBUTE LIST IN THE EXAMPLE ABOVE:

03  THERE ARE THREE CHARACTERS IN THIS ENTRY.
1D  "INTMODE," A DEFAULT ATTRIBUTE.
04  VARIANT OF INTMODE.  04 = "EBCDICV."

03  THERE ARE THREE CHARACTERS IN THIS ENTRY.
08  "KIND" ATTRIBUTE.
09  VARIANT OF KIND.  09 = CARD READER.

03  THERE ARE THREE CHARACTERS IN THIS ENTRY.
1A  "BUFFERS" ATTRIBUTE.
03  NUMBER OF BUFFERS REQUESTED.

03  THERE ARE THREE CHARACTERS IN THIS ENTRY.
OF  "MAXRECSIZE" ATTRIBUTE.
OE  MAXIMUM NUMBER OF WORDS IN 1 RECORD.

00  A ZERO IN THIS POSITION TERMINATES THE LIST.
00
00
00
```

Figure 10-5. Data Pool

```
S          06   6   800000   002800
           05   5   000001   E00000
           04   5   270000   840002
           03   5   270000   740001
           02   1   4FF001   802000
           01   3   000000   0883FC                    TO NORMALEOJ...
D[02]      00   3   CFF001   808002
```

Figure 10-6. Process Stack of FILEEXAMPLE Before Opening Files

Typical write statement code may start out with the following

instructions:

```
MKST

ZERO

NAMC 2,4 (MYDISKFILE)

INDX

LOAD (SELECTOR)
```

On the first pass through this code, an "INVALID OPERAND" interrupt will occur and the hardware will take over by calling the 0,3 procedure. Once in HARDWAREINTERRUPT, it will be determined that the INVALID OPERAND interrupt occurred because we were trying to "INDX" an indexed descriptor with a size field of seven. This being the case, HARDWAREINTERRUPT calls ATTRIBUTEHANDLER. ATTRIBUTEHANDLER is the procedure that processes the file's data pool and builds the FIB and LEB for the file. Before a return is made to the program, the descriptor at 2,4 will have been replaced with a descriptor that points to the newly created FIB.

BUFFER ORGANIZATION

The MCP procedure SETUPTANK is called during file initialization. This procedure is responsible for setting up all IOCB's and buffers as shown in figure 10-7. The first seven words in IOCB's are known to the hardware and must be in the order shown. The remaining words in the IOCB's are known only to the software but will also be in the order given in the figure. ""IOAREA,"" in figure 10-7, represents a data descriptor in the FIB and ""FIB,"" in this figure, is the FIB that the TANK belongs to. The IOAREA word in the FIB, in conjunction with the IOAL words in the IOCB's, is used to ""ROTATE"" the buffers. The BFFREVENT word in IOCB's is the event waited on when a read is performed on an empty buffer, a write on a full buffer, etc. Generally, IOCB's are immediately followed by their buffers and, if possible, a TANK will occupy one contiguous area in memory.

READ STATEMENT CODE

Figure 10-8 contains the in-line code for the read statement at the top of the figure. Read statement code may have many variations but in the example given, the code can be divided into 5 sections. The first section provides entry into the selected I/O procedure. After the I/O has been performed, a ""RETN"" is made to section 2, returning a software result descriptor. If bit 0 is on in this RD, an end-of-file check

is made. If bit 0 is not on, a branch is made to section 5 where the RD is deleted from the top of the stack and the program continues. If an end-of-file check is made and it is the end of the file, then the EOF branch is taken. Otherwise, USERIOERROR is called and the program DS'ed.

Figure 10-7. Buffer Organization

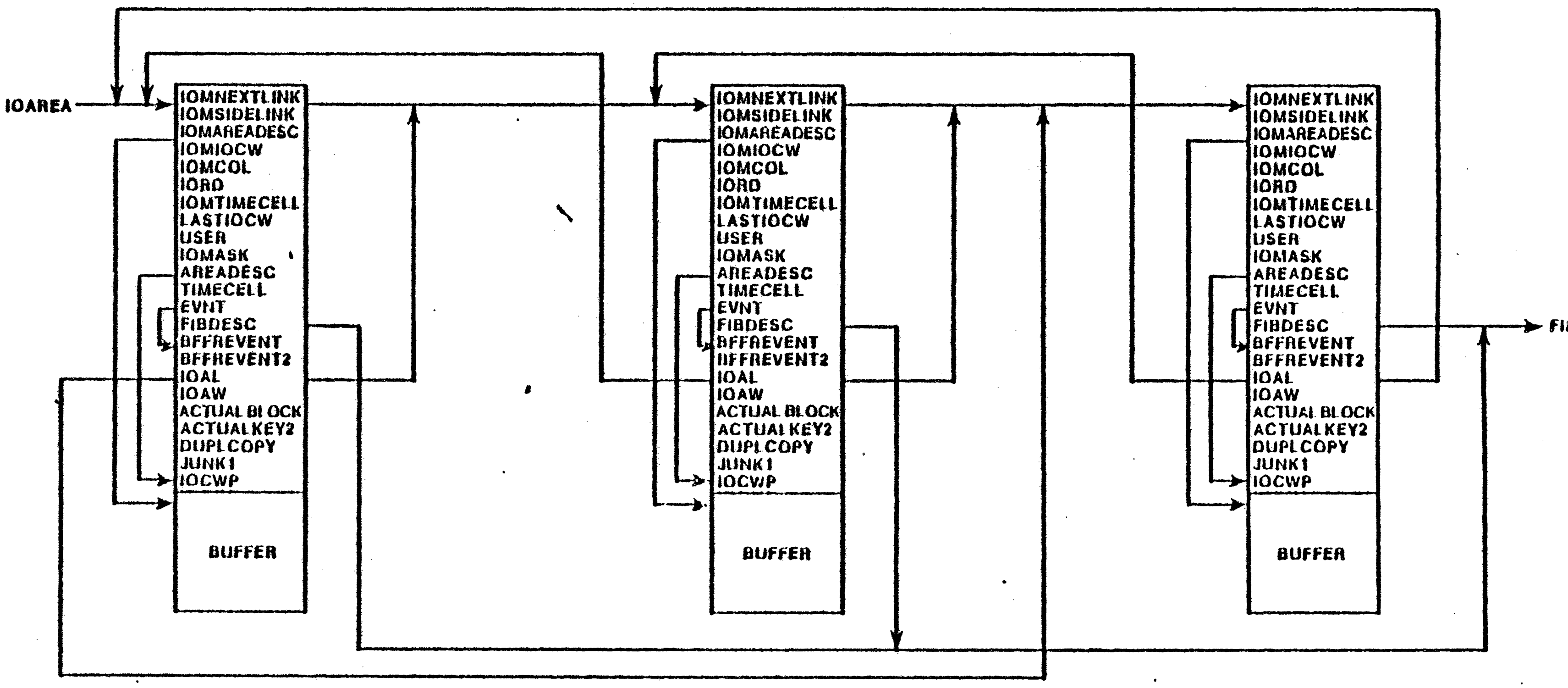


Figure 10-7. Buffer Organization

```
STATEMENT: READ (MYCARDFILE, 14, CARDIMAGE) [CARDEOF];

           MKST
           ZERO
           NAMC 2,3 (MYCARDFILE)
           INDX            (INVALID OPERAND INTERRUPT OCCURS HERE
                            ON FIRST PASS.)
           LOAD
           ONE
           LOR
SECTION 1  ZERO (UNITFEATUE, FIRST PARAMETER)
           LT8 E (SIZE, SECOND PARAMETER)
           ZERO (AREA, THE THIRD PARAMETER)
           NAMC 2,5 (CARDIMAGE)
           INDX
           LT48 080200000180 (CHOOZE, THE FOURTH PARAMETER)
           ENTR

           DUPL
SECTION 2  BRTR      (TO DUPL OPERATOR OF SECTION 3)
           BRUN      (TO SECTION 5)

           DUPL
SECTION 3  ISOL 9:1  (ISOLATE THE END OF FILE BIT)
           BRFL      (TO START OF SECTION 4)
           DLET      (DELETE THE RESULT DESCRIPTOR)
           BRUN      (TO THE END OF FILE LABEL)

           NAMC 0,2E (USERIOERROR)
           EXCH
           IMKS
SECTION 4  NAMC 2,3 (MYCARDFILE)
           STFF
           ENTR

SECTION 5  DLET  (DELETE THE RESULT DESCRIPTOR)
```

Figure 10-8. READ Statement Code From FILEEXAMPLE

UNITFEATURE - CONTENTS DEPEND ON "UFCODEF" IN THE CHOOZE PARAMETER.
EXAMPLE CONTENTS IS RECORD NUMBER FOR RANDOM I/O's.

SIZE - NUMBER OF UNITS (CHARACTERS OR WORDS) TO READ OR WRITE.

AREA - ARRAY USED FOR SOURCE OR DESTINATION OF I/O DATA.

CHOOZE - MISCELLANEOUS INFORMATION.

```
       CALLERTYPEF          = 19:8,

      % 1   = COBOLJOB

      FUNCTIONCODEF         = 11:4,

      % 0   = WRITE
      % 1   = READ

     AFTERF                 =  7:1,

     UFCODEF                =  6:6,


     % 00 = VARRAY
     % 00 = VAEXP
     % 01 = VXALG
     % 02 = VSPACE
     % 03 = VNO
     % 04 = VSKIP
     % 05 = VLINE
     % 06 = VSTATION
     % 07 = VTIME
     % 08 = VSTOP
     % 09 = VSTACKER
     % 10 = VWRITEBLOCK
     % 11 = VBINARYIO

     KEYEDF                 =  0:1,
```

Figure 10-9. Parameters Passed for I/O Operations

I/O PARAMETERS

All I/O procedures are passed four parameters:

UNITFEATURE

SIZE

AREA

CHOOZE

The SIZE parameter gives the number of characters or words to be transferred to/from the AREA parameter, an array. UNITFEATURE's contents depend on the UFCODEF in the CHOOZE parameter and in many cases is not used.

The CHOOZE parameter has five fields:

CALLERTYPEF 19:8

FUNCTIONCODEF 11:4

AFTERF 7:1

UFCODEF 6:6

KEYEDF 0:1

A typical CALLERTYPEF value would be 1, meaning its a "COBOLJOB."" If FUNCTIONCODEF equals 0 it's a write and if it equals 1 then it's a read. Typical UFCODEF values would be:

VARRAY = 0

VXALG = 1

VSPACE = 2

VNO = 3

VSKIP = 4

VLINE = 5

VSTATION = 6

FIB AND LEB

When a FIB or LEB is first seen, the thing that immediately comes to mind is: where to start? Figure 10-10 list all words in the FIB and figure 10-11 list the words in the LEB. Most of the important words in these arrays have already been discussed but for your review, the most important words in the FIB have been picked out and listed below. The most important thing in the LEB is simply the file name and kind.

SELECTOR	BUFFDESC
FIBMSCW	PCW'S
FILESTATUS	PCWCONTROL
IOINF	OFFSET (USED TO INDEX BUFFDESC)

LEB DHEADER

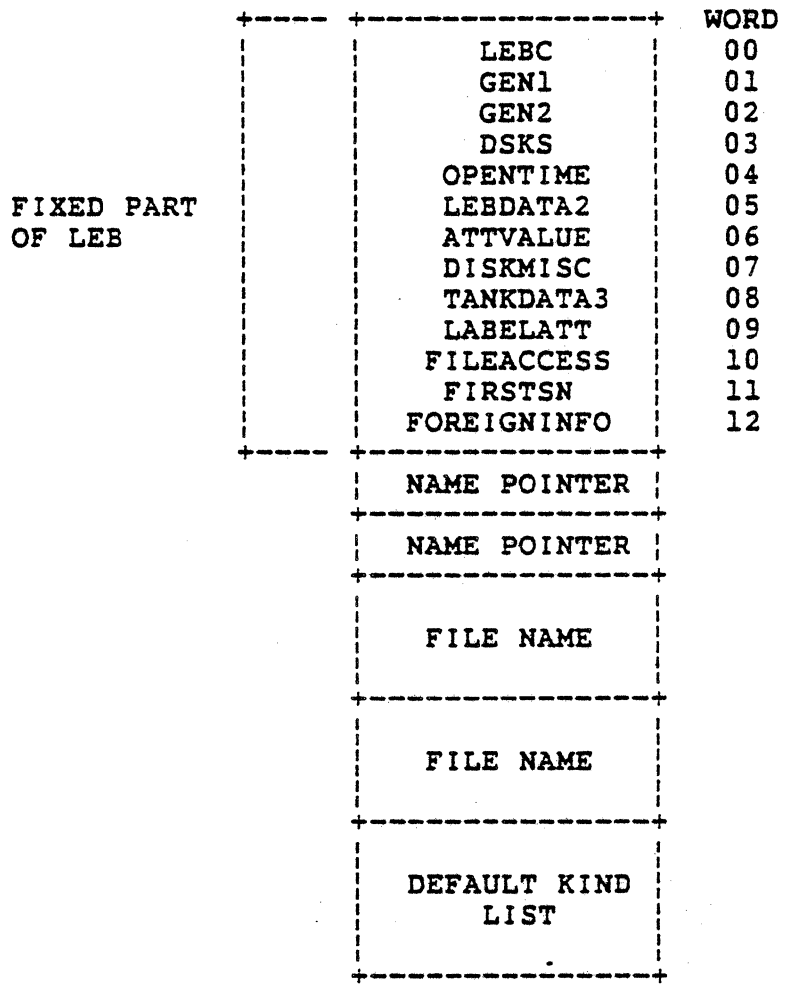
IOAREA

37	TRANSACTIONCOUNT	56
36	IOCB	55
35	FIBLOCKSNR	54
34	FLOPPYMISC	53
33	USEROUTINES	52
32	INPUTTRANSLATION	51
31	OUTPUTTRANSLATION	50
30	FILEEVENT2	49
2F	FILEEVENT1	48
2E	CURRENTBLOCK	47
2D	SOFFSET	46
2C	SIOINFO	45
2B	SBLOCKING	44
2A	ACTNUM	43
29	FIBEOF	42
28	DHEADER	41
27	AEXP	40
26	T	39
25	I	38
24	RECSIZE	37
23	MINRECSZ	36
22	UPPER	35
21	LOWER	34
20	BLOCKCOUNT	33
1F	RECORDCOUNT	32
1E	OFFSET	31

Figure 10-10. File Information Block (1 of 2)

```
          1D    PCWCONTROL          30
          1C    PFLOAT              29
          1B    PWAIT               28
          1A    PMOVEIN             27
          19    PMOVEOUT            26
          18    PRELEASE            25
          17    PLOCKER             24
          16    PSEARCH             23
          15    PREADR              22
          14    PSEEK               21
          13    PREADN              20
          12    PWRITEN             19
          11    PREADS              18
          10    PWRITES             17
          0F    FILIOTIME           16
          0E    FMTLOCK             15
          0D    FMTBUFFDESC         14
          0C    SIOAREA             13
          0B    BUFFDESC            12
          0A    IOAREA              11
          09    LEB                 10
          08    IOINFO              09
          07    PAGESPEC            08
          06    DISKBLOCK           07
          05    TANKDATA2           06
          04    TANKDATA1           05
          03    FILESTATUS          04
          02    RECORDSTATUS        03
          01    FIBLOCK             02
D[1] =>   00    FIBMSCW             01
                SELECTOR            00          MYCARDFILE
```

. Figure 10-10. File Information Block (2 of 2)



The Label Equation Block is declared in the logical I/O section of the MCP. The fields within LEB words plus other information pertaining to LEB's may be found here.

Figure 10-11. Label Equation Block

IOPCWS

As previously discussed, the IOPCWS array is set up by FIBSTACK during system initialization. Due to the sequence in which the FIBSTACK procedures are declared, the IOPCWS array may be broken down into 5 sections as shown in figure 10-12. The "MISC. PROC." section consists of COLUMN PCW's such as OPEN, CLOSE, ILLEGALCOLUMNACTION, etc. The second section in IOPCWS contains the RANDOM diagonals, the third section contains serial DIAGONALS. The fourth section contains PCW's for procedures that handle I/O requests such as "WRITE[NO]" and "READ[NO]." These PCW's will be found in the PMOVEIN (for reads) and PMOVEOUT (for writes) FIB locations. The last section in IOPCWS contains DIAGONAL PCW's for the reverse read procedures.

The PCW's in the IOPCWS array are referenced simply by number in the first section and by a define in the serial and random reads and writes and reverse reads. This define references the PCWCONTROL in the associated FIB.

PCWCONTROL is set up at file open time in the procedure SETUPFIB. This word contains three fields and is referenced as follows:

POLOC = PCWCONTROL.[15:16]

Always points to a serial write PCW.

P1LOC = POLOC + 1

Always points to a serial read PCW.

P2LOC = PCWCONTROL.[31:16]

Always points to a random write PCW.

P3LOC = P2LOC + 1

Always points to a random read PCW.

P4LOC = PCWCONTROL.[47:16]

Always points to a reverse read PCW.

POLOC will give an index into IOPCWS to the correct DIAGONAL PCW for a particular type of serial write and since the associated read PCW will immediately follow, P1LOC will point to the correct serial read PCW. This idea also applies to P2LOC and P3LOC. P4LOC points to the correct reverse read PCW, there being no reverse writes.

I/O PROCEDURES FLOW

This is the last sub-section in this manual and consists of a detailed discussion of the I/O procedures involved in a typical I/O operation. While reading this sub-section, it would be appropriate to relate this discussion to the program of figure 10-4 and the FIBSTACK section of the MCP source listing. These will not be directly referenced but the outlines at the end of this section will be.

This discussion starts with the assumption that a card file is to be read for the first time. The following are the file declaration and read statement to be used:

```
DECLARATION:  FILE MYCARDFILE (KIND = READER,
                               BUFFERS = 3,
```

```
                                    MAXRECSIZE = 14);

STATEMENT: READ (MYCARDFILE, 14, CARDIMAGE);
```

When the file is first referenced, a hardware interrupt will occur and eventually ATTRIBUTEHANDLER will be called. ATTRIBUTEHANDLER will set up the FIB and LEB as previously discussed (figure 10-2) and then exit back to the program where a second attempt will be made to enter the PREADS location in the FIB. The PCW in the PREADS location at this time will be READSCOLUMN (and the FILESTATE will be 0).

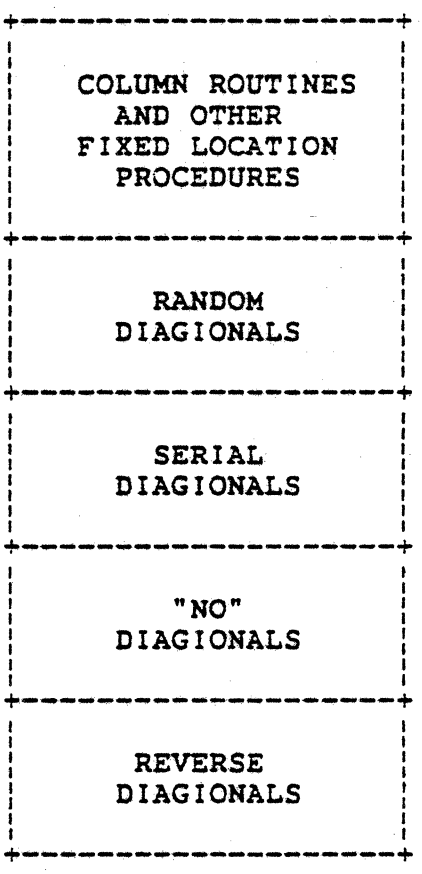


Figure 10-12. IOPCWS Array Layout

Before continuing with the discussion on READSCOLUMN, let's briefly discuss the PCW's in the FIB at this time.

```
PWRITES ILLEGALCOLUMNACTION

PREADS READSCOLUMN

PWRITEN ILLEGALCOLUMNACTION

PREADN READNCOLUMN

PSEEK SEEKCOLUMN

PREADR ILLEGALCOLUMNACTION

PSEARCH SEEKNONDISK

PLOCKER LOCKCODE

PRELEASE RELEASESIMPLE

PMOVEOUT NOWRITEUW

PMOVEIN NOREADUW

PWAIT WAITSIMPLE

PFLOAT OPEN
```

The above list is the default set up for the FIB on card files. Some locations will never be referenced and some (ILLEGALCOLUMNACTION for instance) will result in the program being DS'ed. All PCW's in the list came from the IOPCWS array and were placed in the FIB by ATTRIBUTEHANDLER.

The READSCOLUMN procedure is responsible for setting up the TANK shown in figure 10-2, filling the buffers, and in this case, transferring the first card's information to the CARDIMAGE array. Also, READSCOLUMN will place a PCW to SERIALREADUW in the PREADS location, overwriting its own PCW. The ""UW"" here stands for ""UNBLOCKED WORDS.""

In reference to the READSCOLUMN outline at the end of this section, it can be seen that the procedure starts with a CASE on FILESTATE. Since FILESTATE is 0 at this time, CASE 0 will be executed which results in a call on the FIBSTACK OPEN procedure. This procedure's PCW is currently in the PFLOAT FIB location. OPEN has the responsibility of finding the file and calling SETUPTANK to set up the IOCB's and buffers. OPEN also sets the FILESTATE to 1.

Returning from OPEN, READSCOLUMN continues with a call to PREADS. READSCOLUMN is calling itself but with a FILESTATE of 1. CASE 1 is sometimes referred to as the SECOND OPEN. This

CASE sets the FILESTATE to 3 (READSTATE) and calls SETUPFIB which inserts appropriate PCW's in the PRELEASE, PWAIT, PMOVEIN and PMOVEOUT locations and also sets up the PCWCONTROL word in the FIB. PCWCONTROL has three fields, one pointing to the correct SERIAL DIAGONAL, another pointing to the correct RANDOM DIAGONAL and the last pointing to the correct REVERSE READ DIAGONAL. The last thing SETUPFIB does is to fill the buffers and then return to READSCOLUMN.

At this point in READSCOLUMN's execution, the CASE statement is left and the following statement is executed:

```
PREADS := IOPCWS[P1LOC];
```

This is the statement responsible for replacing the COLUMN PCW in PREADS with the correct DIAGONAL PCW. If you recall, P1LOC is a define referencing the PCWCONTROL word in the FIB.

The last thing done in READSCOLUMN is to go to BINARYIOSTARTER which will transfer the first buffer to the CARDIMAGE array, start a new I/O to fill the empty buffer and return a software R/D to the calling program. BINARYIOSTARTER makes use of the PMOVEIN location (containing the NOREADUW PCW at this time) to perform the core-to-core transfer of the buffer to the CARDIMAGE array.

On all subsequent entries on the PREADS location, READSCOLUMN WILL NOT BE ENTERED. There is a DIAGONAL PCW at this location now and it will not be changed unless the file changes state (which is highly unlikely in this example). The DIAGONAL PCW in the PREADS location is SERIALREADUW and its outline is at the end of this section. The DIAGONAL procedures are so short and simple that this outline is nearly a line for line copy of the procedure in the MCP source listing.

There are three comments to be made in reference to the SERIALREADUW procedure. One is in reference to "WAITONBUFFER". This is a call on a PCW in the PWAIT FIB location. WAITSIMPLE is currently in this location. WAITSIMPLE checks the HAPPENED BIT in IOAREA[BUFFEREVENT] and if it's set, the buffer contains a valid record and an exit is made back to the calling procedure. If the HAPPENED BIT is reset, WAITSIMPLE will call the GLOBAL MCP procedure WAITP passing the BUFFEREVENT.

Returning from WAITSIMPLE, we are guaranteed the buffer contains a valid record (pointed to by the FIB BUFFDESC word) and a transfer is made with a REPLACE statement (the second point of interest).

The third comment in reference to the SERIALREADUW outline is about the "RELEASEBUFFER" statement. This statement is a call on the PRELEASE FIB location which at present contains a PCW to RELEASESIMPLE.

RELEASESIMPLE is the procedure that resets the HAPPENED BIT in the IOCB BUFFEREVENT word and calls IOREQUEST to link the IOCB into the I/O queue and initiate the queue (START I/O). It is up to the MCP procedure, IOFINISH, when processing the IOM's RESULTQ, to cause the IOCB's BUFFEREVENT and wake up any tasks that were waiting on the I/O to complete.

OUTLINE OF ATTRIBUTEHANDLER

1. If the FIB is not allocated:
   a. Get memory for FIB. Put SELECTOR and FIBMSCW in FIB.
   b. Read in data pool for file.
   c. Get memory for LEB.
   d. Transfer file name to LEB.
   e. Put column PCW's in FIB PCW locations.
2. If this is an IPC FIB it must be locked.
3. Put the FIB descriptor in the stack in place of the data pool descriptor.
4. The Label Equation for the file is found.
5. The attributes from the data pool and file label equation are placed in the FIB. This is done in a case statement based on attribute number.
6. If FIB was locked, it is unlocked.

OUTLINE OF READSCOLUMN

1. This column routine is a case statement based on FILESTATE.
2. The NOTOPENED case will call procedure OPEN and then call the PREADS PCW.
3. The NULLSTATE is the second open. The FILESTATE is set to READSSTATE. INITIALIZEFIB (SETUPFIB) is called. The PREADS PCW is called.
4. The other cases involve changing from one state to another. To do this the old state's PCW must be replaced with a column routine. The PREADS PCW must be replaced with a diagonal routine. This routine is then called.

OUTLINE OF OPEN
------- -- ----

1. Call FINDOUTPUT or FINDINPUT to associate the logical file with a physical file.

2. A case statement is executed based on the unit type of the file. These cases will handle the details of each unit.

3. An OPEN record is written to the system log.

4. Call SETUPTANK to get buffers for the file.

5. Set FILESTATE to NULLSTATE.

OUTLINE OF SETUPFIB
------- -- --------

1. Select diagonal routines for the FIB. The index values (into IOPCW's) are saved in PCWCONTROL.

2. Set up RELEASE and WAIT routines in FIB.

OUTLINE OF RELEASESIMPLE
------- -- -------------

1. Call IOREQUEST to do physical I/O.

2. Rotate buffers so program will have a buffer to fill.

OUTLINE OF BINARYIOSTARTER
------- -- ---------------

1. Move data into or out of user's area. It is placed in or comes from the file's buffer. This is done by calling PMOVEIN or PMOVEOUT.

2. The buffer may be released.

OUTLINE OF NOREADW
------- -- -------

1. This is code used by PMOVEIN.

2. Wait for buffer I/O to complete. Procedure at PWAIT is called.

3. Move data from buffer to user's area.

OUTLINE OF SERIALREADUW

1. Wait for buffer I/O to complete.
2. Move data from user's area to buffer.
3. Release buffer.

OUTLINE OF WAITSIMPLE

1. If BUFFEREVENT has not happened, wait on BUFFEREVENT.

OUTLINE OF CLOSE

1. Change PCW's to column routines.
2. Set FILESTATE to NULLSTATE.
3. Execute case statement based on type of unit being closed.
4. Release buffers.
5. Write log record.

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS