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 3.4

REVISION DATE: FEBRUARY, 1985

The within information is not intended to be nor should such be construed as an affirmation of fact, representation or warranty by Burroughs Corporation of any type, kind or character. Any product and related materials disclosed herein is only furnished pursuant and subject to the terms and conditions of a duly executed license agreement. The only warranties made by Burroughs with respect to the products described in this material are set forth in the above mentioned agreement.

The names, places and/or events depicted herein are not intended to correspond to any individual, group or association existing, living or otherwise and are purely fictional. Any similarity or likeness of the names, places and/or events with the names of any individual, group or association existing or otherwise is purely coincidental and unintentional.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

PREFACE

This manual was written to be used in conjunction with the large systems MCP course. Care has been taken to explain, to some degree, all subjects covered in the MCP course. 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 BASIC SYSTEM SUPPORT course.
3. Currently, quite familiar with ALGOL and the Large Systems hardware.

SECTION 1

HARDWARE OVERVIEW

INTRODUCTION

This section is an overview of Burroughs large systems hardware. Because of hardware differences among large systems the hardware overview is divided into five sections as follows:

B7800 hardware

B7900 hardware

B5900 hardware

B6900 hardware

B5900/B6900 I/O Overview

B7800 HARDWARE OVERVIEW

GENERAL

The Burroughs B7800 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 (CPM), Input/Output Modules (IOM), and Memory Control Modules (MCM) on an electronic grid, or exchange (figure 1-1). The MCP can be changed dramatically by merely setting or resetting run time options with simple operator input commands. The system may be balanced by other operator instructions (and dynamically by the MCP) that control the interaction of the independently operating CPM's, IOM's, and MCM's. 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 scheduling jobs in the multiprogramming mix. The MCP 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 (error detection and correction, redundancy of power supplies) exclude faulty modules from the system and permit processing to continue (without reprogramming) with a temporarily reduced configuration.

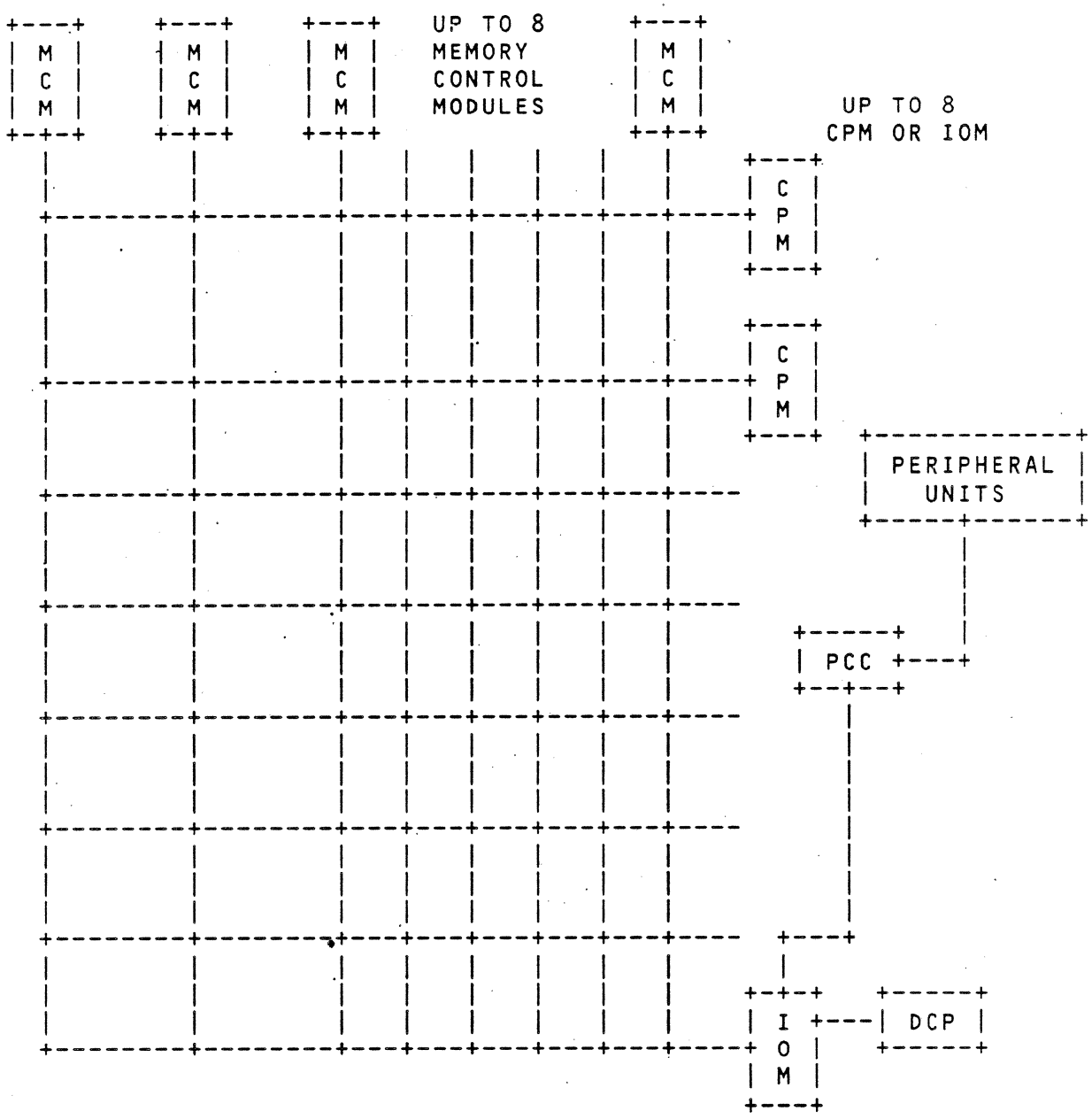


Figure 1-1. B7800 Exchange

SYSTEM CONFIGURATION

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

1. Central components of the system: Central Processor Module (CPM), Input/Output Module (IOM), Memory Control Module (MCM), Maintenance Diagnostic Processor (MDP), and operator's console.

2. Standard Burroughs cabinets that contain peripheral controls and exchanges, Data Communications Processor (DCP), and AC power supplies.

3. Peripheral devices that are joined to the central system by means of peripheral controls and exchanges. 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 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 system which is composed of Central Processor Modules (CPM), Input/Output Modules (IOM), Memory Control Modules (MCM) and Data Communication Processors (DCP).

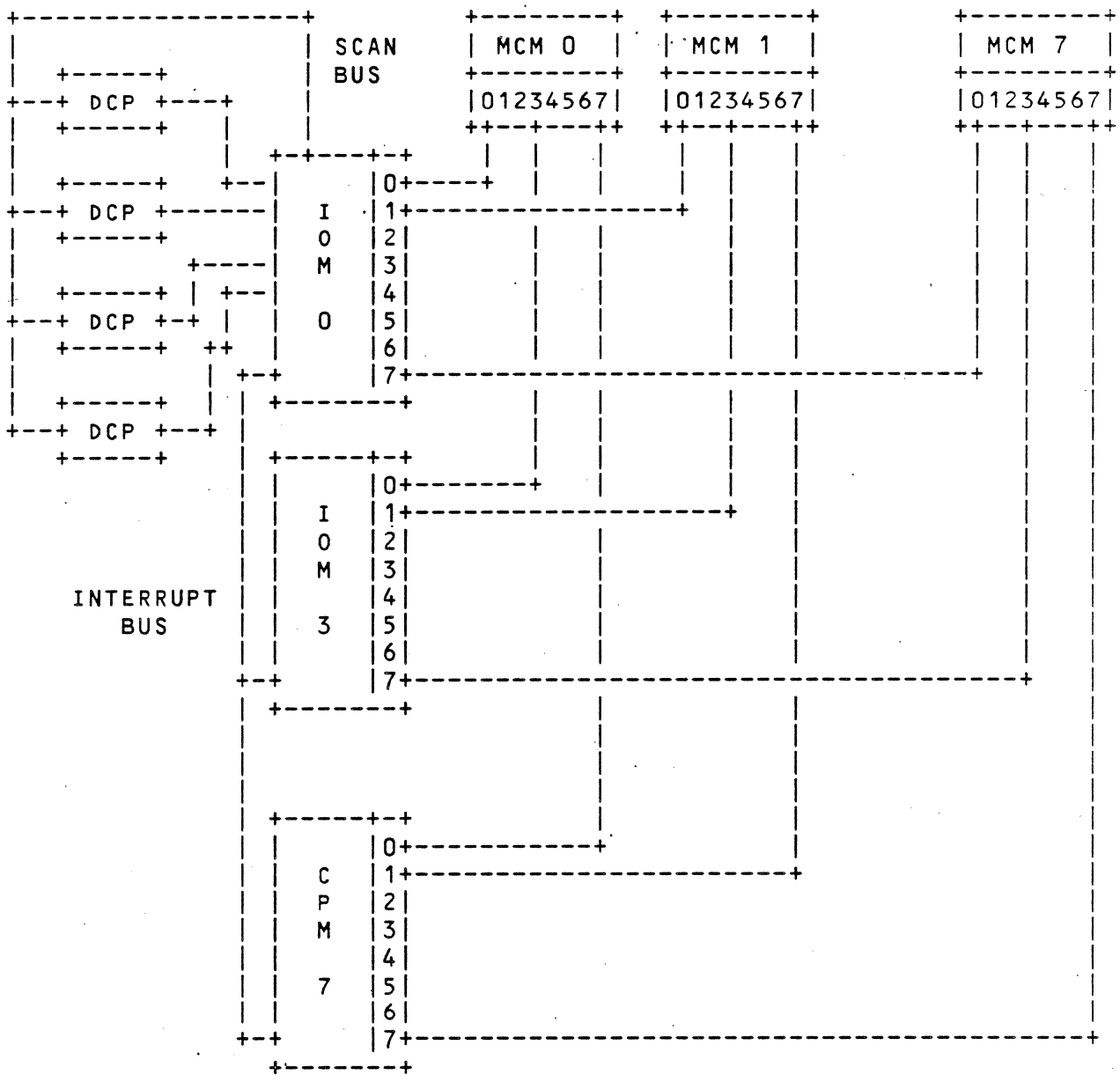


Figure 1-2. B7800 SYSTEM

The following general comments can be made about the 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. An 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 a 2K word data buffer and a 2K word code buffer. There is 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 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.

MDP

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 CPM or with the Maintenance Processor (MP). The MP is a B800 system. Thus, maintenance operations can be run by a 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

Memory Control Modules (MCM's) form the interface between a requestor (IOM and CPM) and memory. Each requestor is connected to each MCM. A system can be configured with up to 8 MCM's. The MCM is connected to memory storage cabinets which contain actual memory storage (planar or IC memory).

Figure 1-3 shows two MCMs with planar memory. 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 in one operation 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 MCM with IC memory. Each MCM can have up to 2 Memory Storage Cabinets (MSC). A MSC can have up to four Memory Storage Units (MSU). Each MSU has 131K words. Memory is phased at the MSU level and will phase at most 8 words. Thus, the maximum number of words that can be transfered in one operation 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

There are two models of MCM's (model II and model III). A model II MCM can control 256K words of memory. A model III MCM can control 1 million words of memory. This configuration (1 MCM with 1 million words) would provide only one path to memory and could cause performance problems. Better performance could be achieved by using two model III MCM's with each MCM controlling 500K words.

A requestor can address 1 million words. Therefore a monolithic system can use at most 1 million words. More memory can be configured but would not be usable. Any additional memory could be used if the system was split into two systems. A system running tightly coupled can use up to 2 million words (2.5 million words using model III MCM's). This tightly coupled configuration (2.5 million words) would require 5 model III MCM's, 4 CPM's and 4 IOM's.

Words stored in memory are 60 bits as follows:

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 (memory phasing).

Locking out MCM's to requestors. This is used when the 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.

One should note a requestor (not the MCM) 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 and make the request to the MCM again.

As an example of how these switches are set, assume a system is configured as follows: (see figure 1-3)

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

The switches should be set as follows:

MCM	LOWER	UPPER	INHIBIT	MSU STATUS
1	000000	001111	11011101	1111
4	010000	011111	11011101	1111

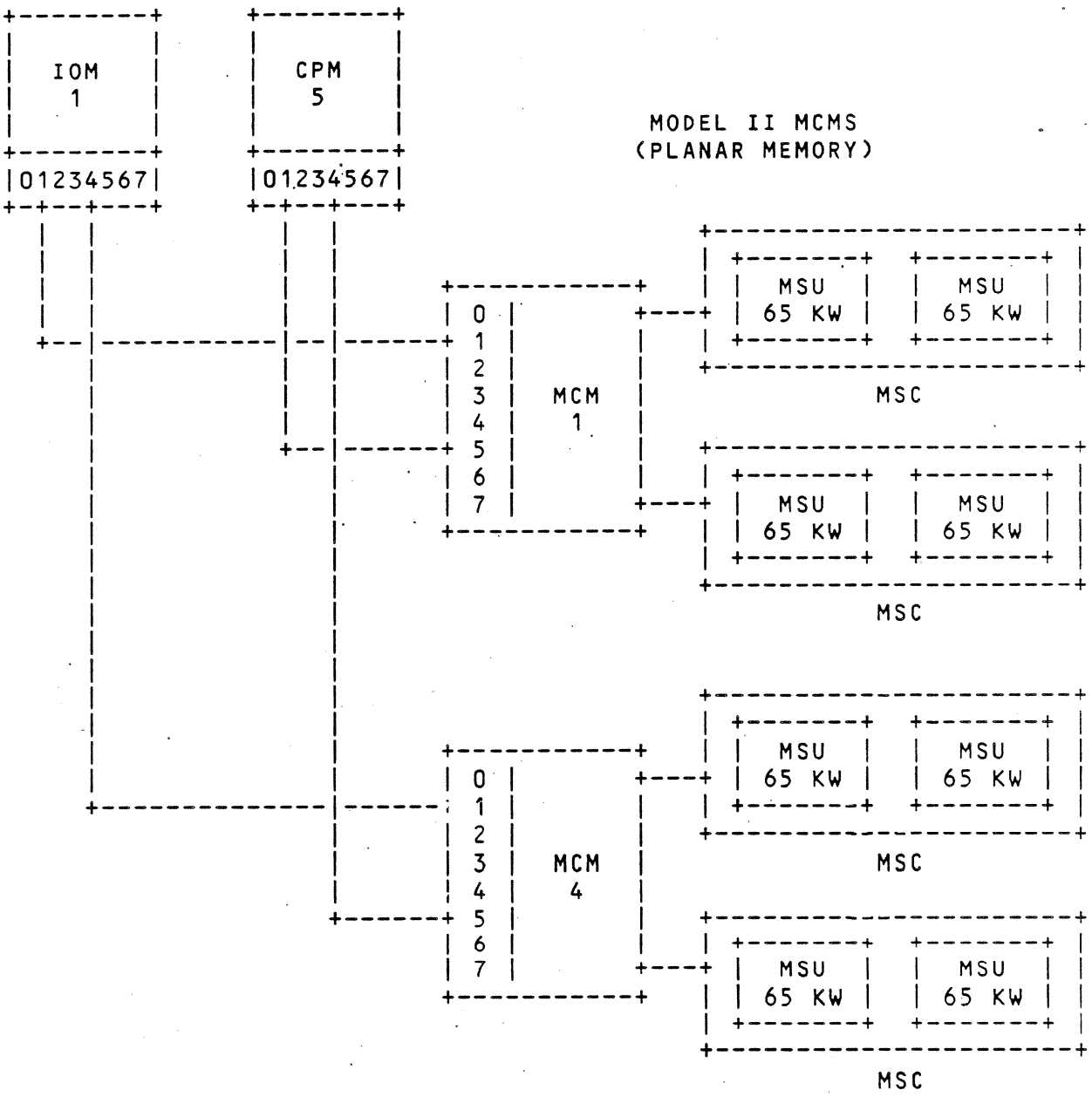


Figure 1-3. MCM Configuration and Planar Memory

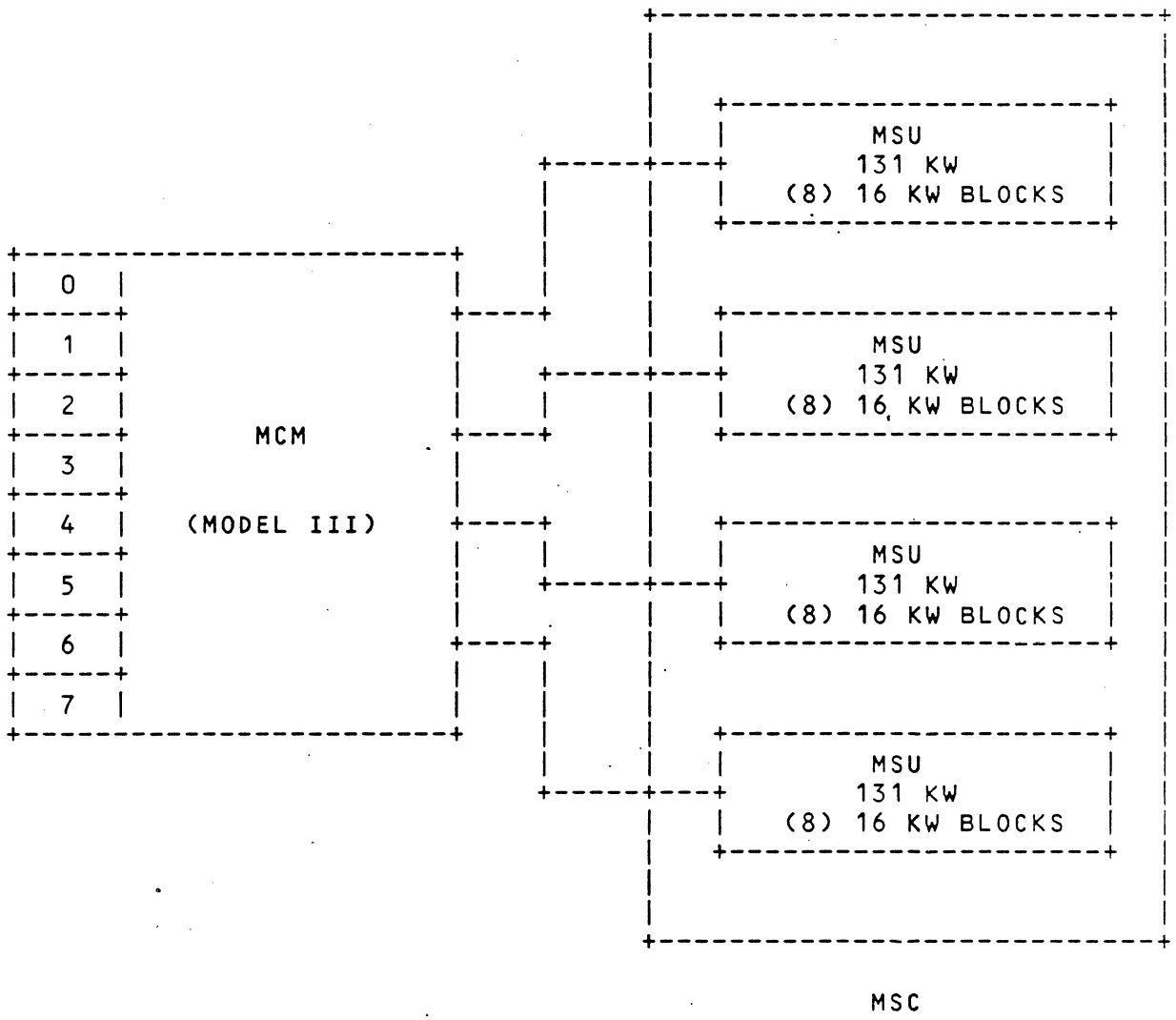


Figure 1-4. MODEL III MCM Block Diagram (IC Memory)

CPM

The CPM is composed of several independent asynchronous units. It is this independence that allows data fetches to be overlapped with instruction execution. The basic functions of these units in the CPM are described in the following paragraphs (see figure 1-5).

MEMORY ACCESS UNIT (MAU):

The memory access unit provides the memory interface for the CPM. The MAU is capable of doing simultaneous fetches and stores to memory as long as the addresses are not in the same MCM.

PROGRAM CONTROL UNIT (PCU):

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, the PCU is responsible for allocation of registers in the Central Data Buffers (CDB).

EXECUTION UNIT (EU):

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

The EU includes logic for short arithmetics, add operations less than 20 bits and multiply operations with results less than 16 bits.

DATA REFERENCE UNIT (DRU):

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

STORE QUEUE (SQ):

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 (ASSOCIATIVE MEMORY):

The program buffer will hold up to 2K words of object code in the CPM. The data buffer will hold up to 2K words of data and control information.

CENTRAL DATA BUFFER (CDB):

The central data buffer (64 words) 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 an EGG TIMER interrupt is generated. This will prevent the CPM from looping in control state.

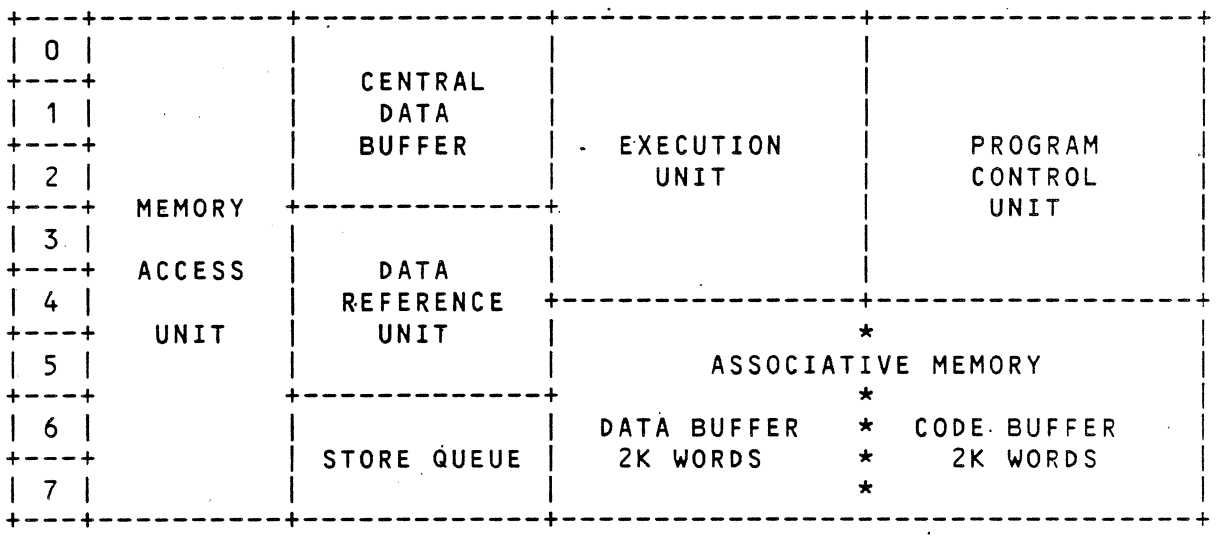


Figure 1-5. CPM Block Diagram

The following discussion on associative memory is based on the B7800 CPM.

Figure 1-6 is a diagram of how the code buffer is set up. Three data structures are used. These structures are the address array, priority array and the actual code buffers.

To access a word in this part of associative memory bits [7:6] of the 20 bit address are used to get the block number. This block number is used to index the address array. By comparing bits [19:12] of the address with the eight groups in the block you can determine if the word you are looking for is present.

Assuming the address is present, bit [0:1] selects the EVEN or ODD buffers. The group number (determined in the address array) is used to choose a 128 word buffer. Bits [7:7] of the address are used to index into the 128 words.

The priority array has a three bit entry for each of the 64 blocks. It contains the next group number to be overlayed. When that group is overlayed the array value is incremented by one. When seven (111) is reached it will go back to zero (000).

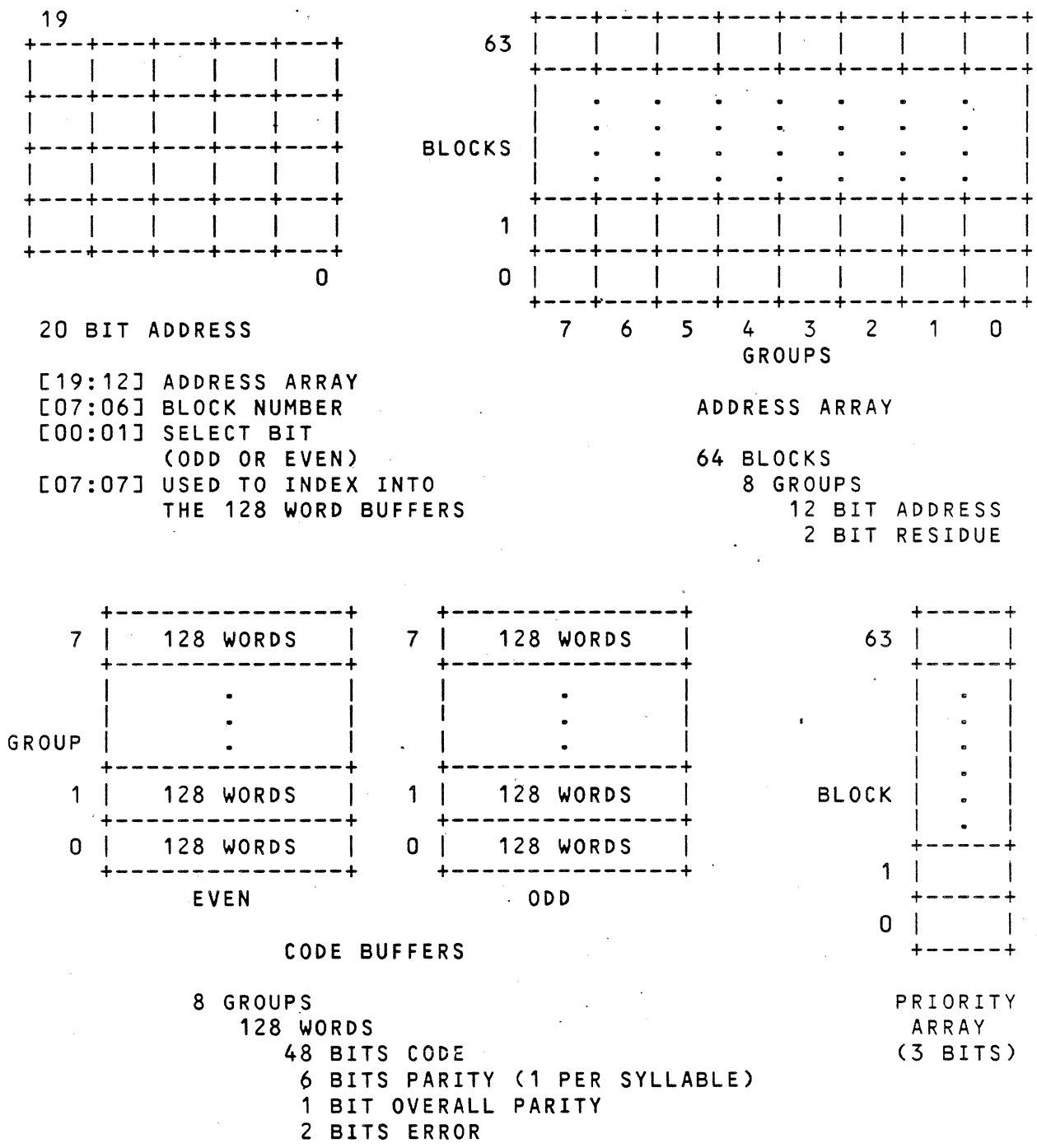


Figure 1-6. CODE BUFFER (Associative Memory)

Figure 1-7 is a diagram of how the data buffer is set up. Again three data structures are used. These include the address array, the priority array and the data array.

To access a word in this part of associative memory bits [8:6] of the 20 bit address are used to get the block number. As with the code buffer, this block is indexed in the address array. Bits [19:11] of the address are compared with the four groups to see if the address is present.

Assuming the address is present, the block number and the group number are used to locate the proper eight word group in the data array. Bits [2:3] of the address are used to get the proper word from the eight word group.

The priority array has 64 entries, 5 bits each. Two bits indicate the 'oldest' group in the block. Two bits indicate the 'second oldest' group in the block. The fifth bit determines the order of the other two groups. The fifth bit is set if the 'newest' group (number) has a greater value than the 'second newest' group (number).

The priority array is updated whenever new data is written to the data buffer and also whenever a word local to the buffer is accessed. If new data is to be loaded, the 'oldest' group entry is overlayed. If a word already local in the buffer is accessed it is moved to the 'newest' position in the priority array.

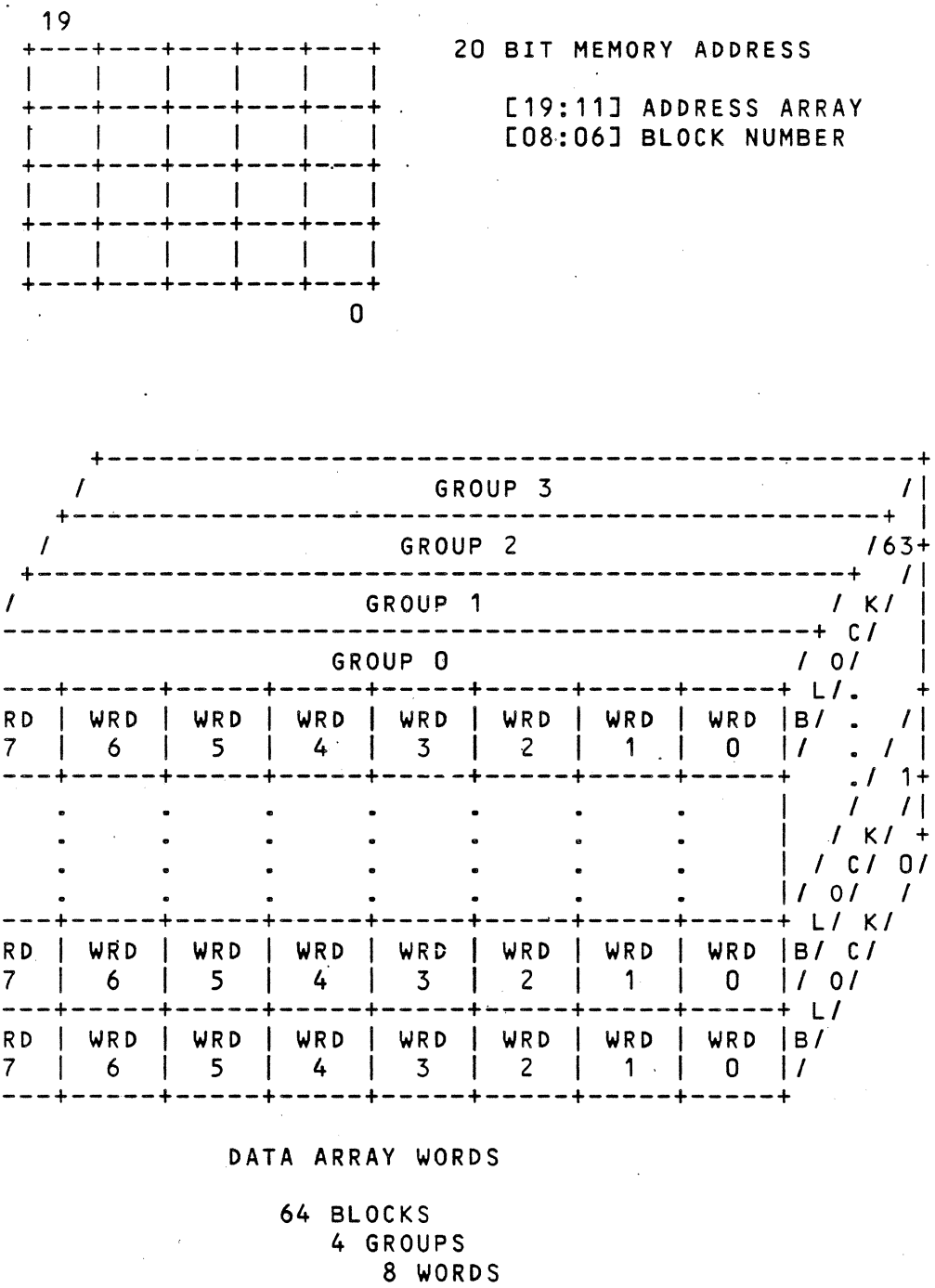


Figure 1-7. DATA BUFFER (Associative Memory) (1 of 2)

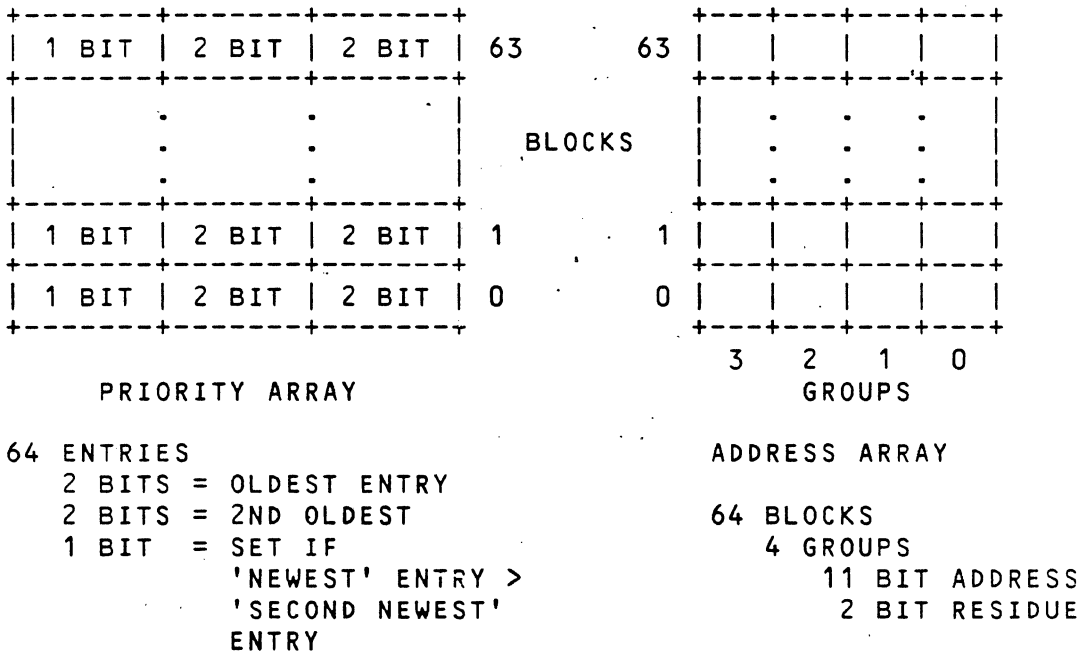


Figure 1-7. DATA BUFFER (Associative Memory) (2 of 2)

IOM

The IOM serves as a buffer and control unit for all I/O operations. The IOM is composed of several asynchronous units. These units can be seen in figure 1-8 and are described in the following paragraphs.

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 the IOM and DCP'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" (DFIB). 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 (PC) bus is used to connect the IOM to the Peripheral Control Cabinet (PCC). The PCC contains peripheral controls. Data is transferred to the IOM in one or two byte groups. The PCC is independently powered or powered by B6700 style AC mods.

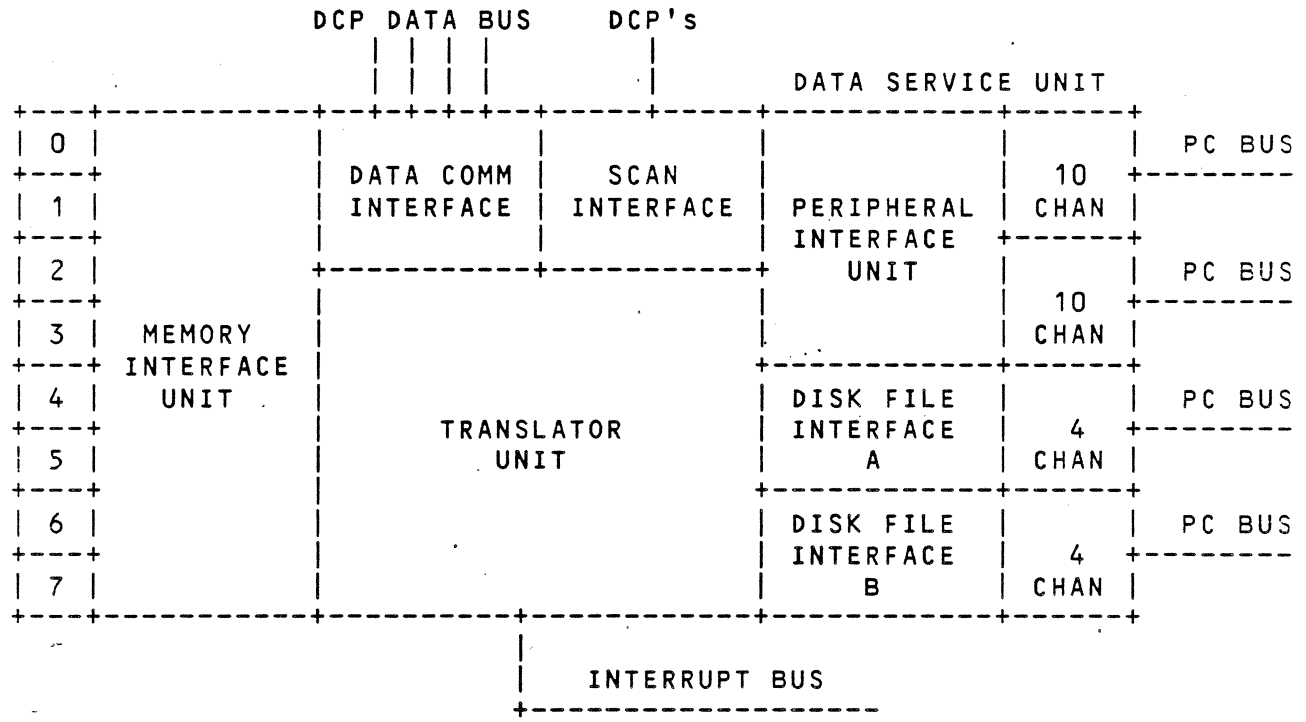


Figure 1-8. IOM Block Diagram

The IOM and MCP share several data structures. These structures are located in main memory and are shown in figure 1-9. The following paragraphs describe 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 operator can decide when it wants the MCP to be interrupted by the IOM. This interrupt will cause the MCP to look at the result queue. The system can be set up (see SBP ODT command) so the IOM will interrupt the MCP when:

An I/O queue is empty or CPM is idle
Each I/O finishes
A task is waiting on an I/O or CPM is idle
The 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 and are defined as follows:

HA[0]
Command to IOM.

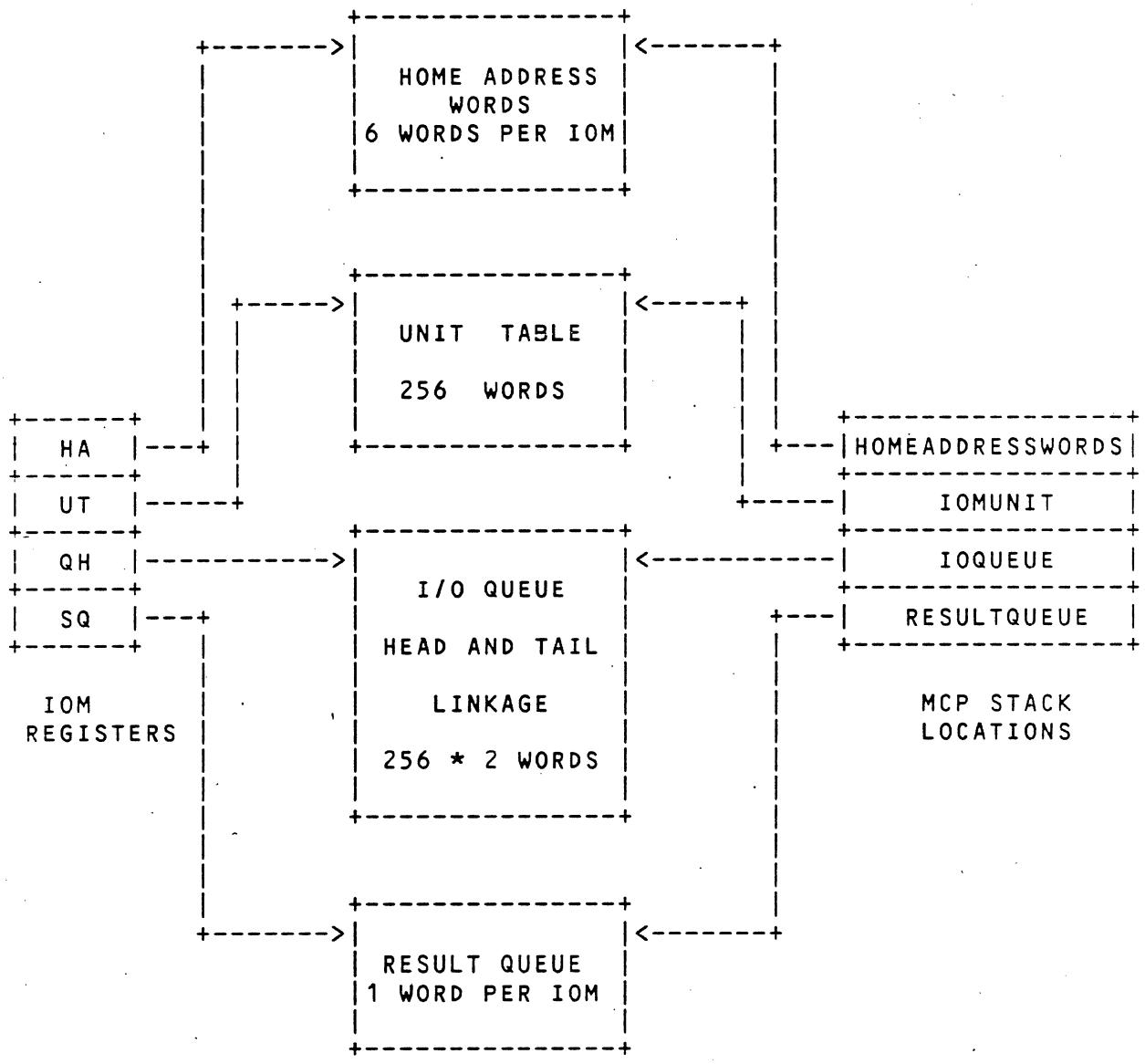
HA[1]
Special information (peripheral status).

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's

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)
   BE SURE LAST COMMAND IS CLEAR
   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 .5 sec. If the command is still not clear the CPM will interrupt the IOM again. If the command is still not clear after .5 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.

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.

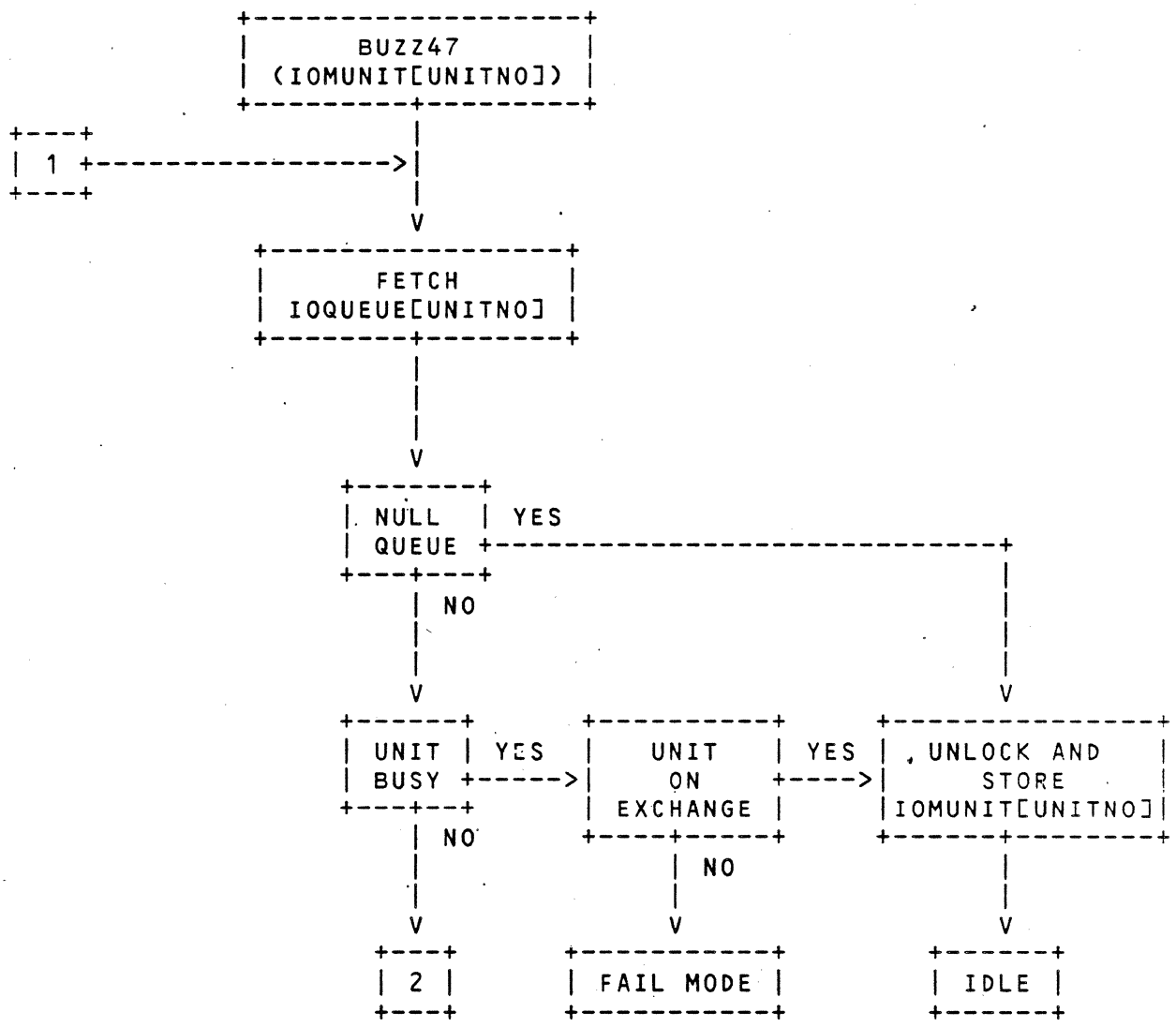


Figure 1-10. START I/O (PAGE 1 OF 2)

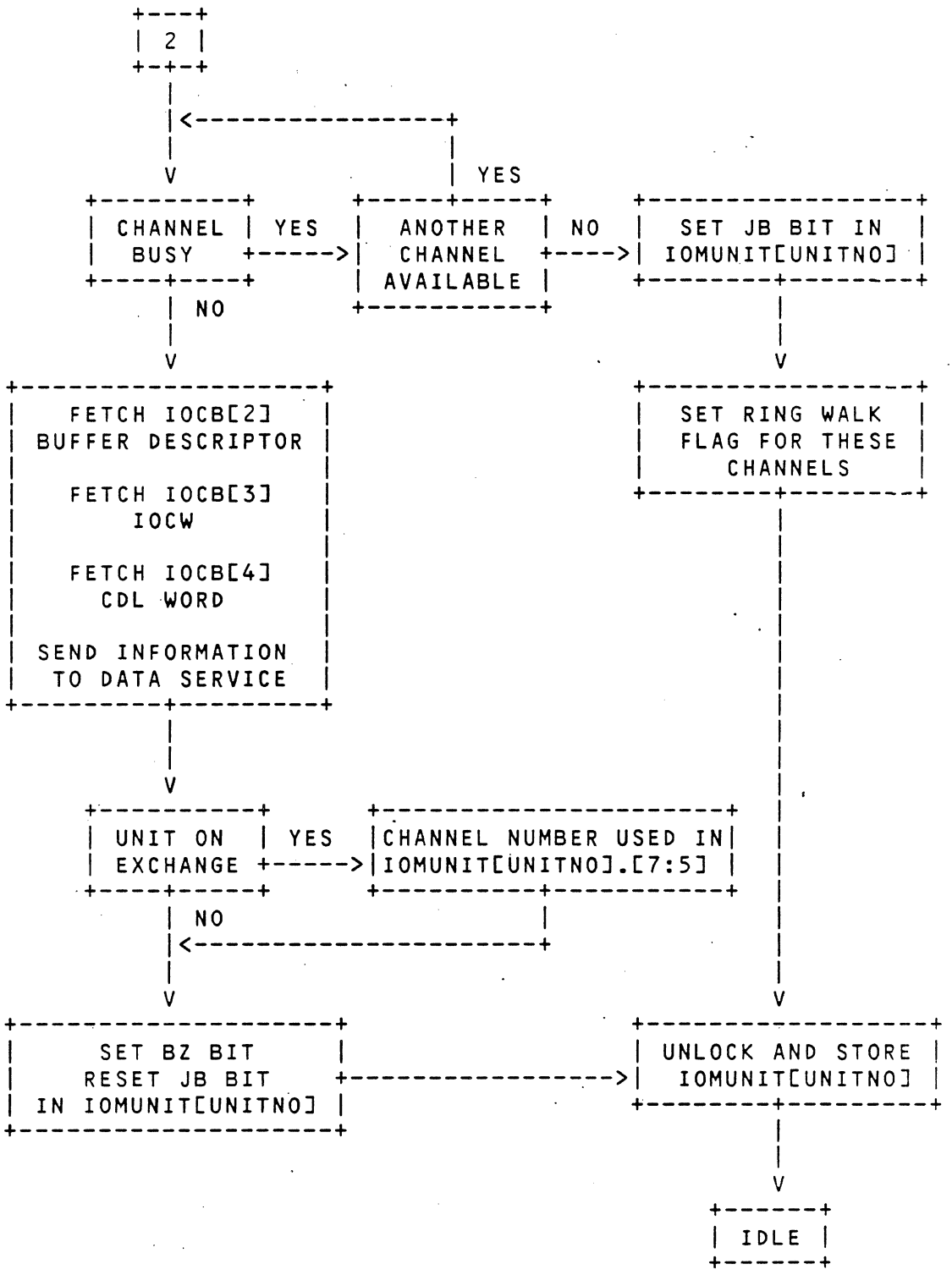


Figure 1-11. START I/O (PAGE 2 OF 2)

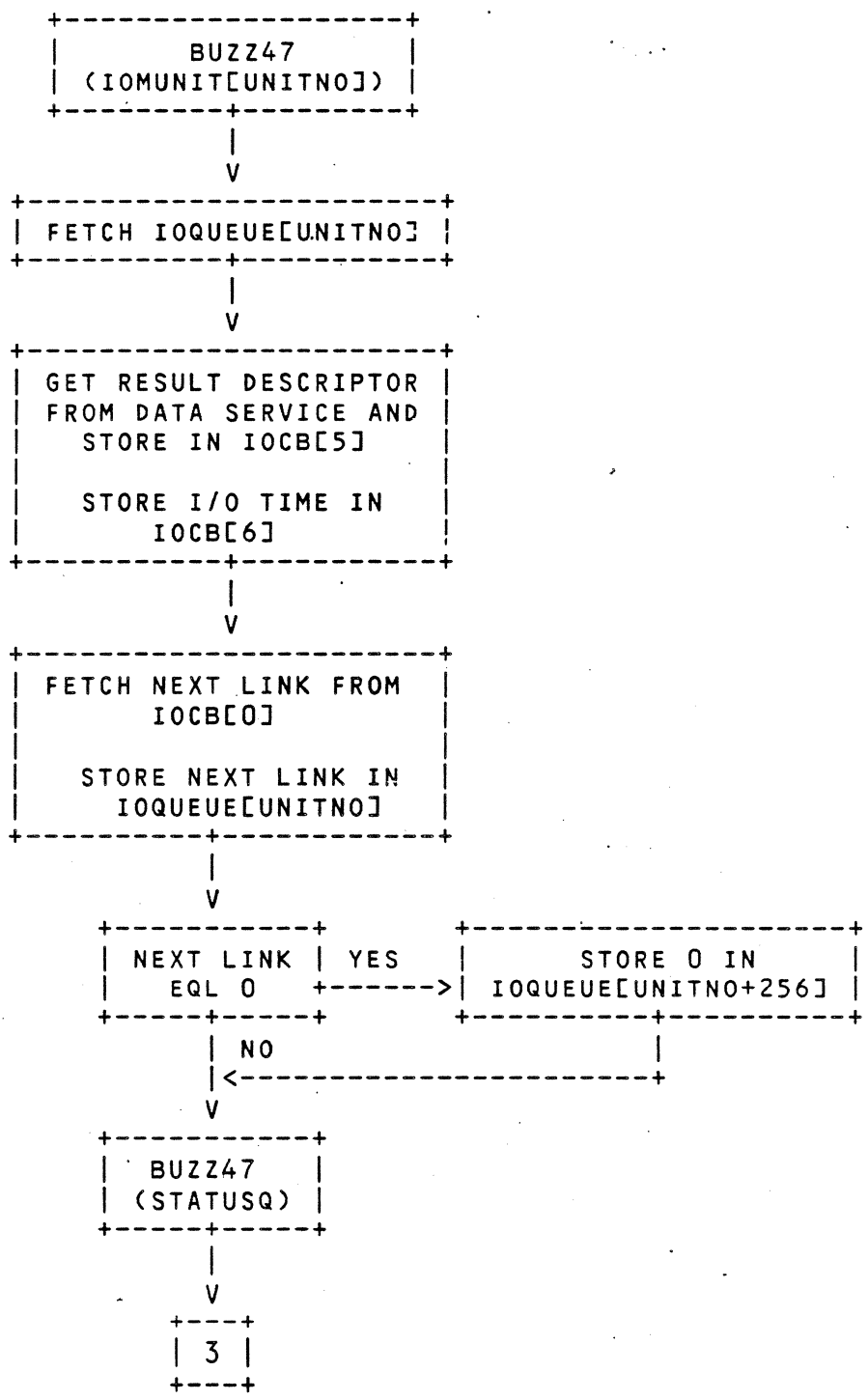


Figure 1-12. TERMINATE I/O (PAGE 1 OF 3)

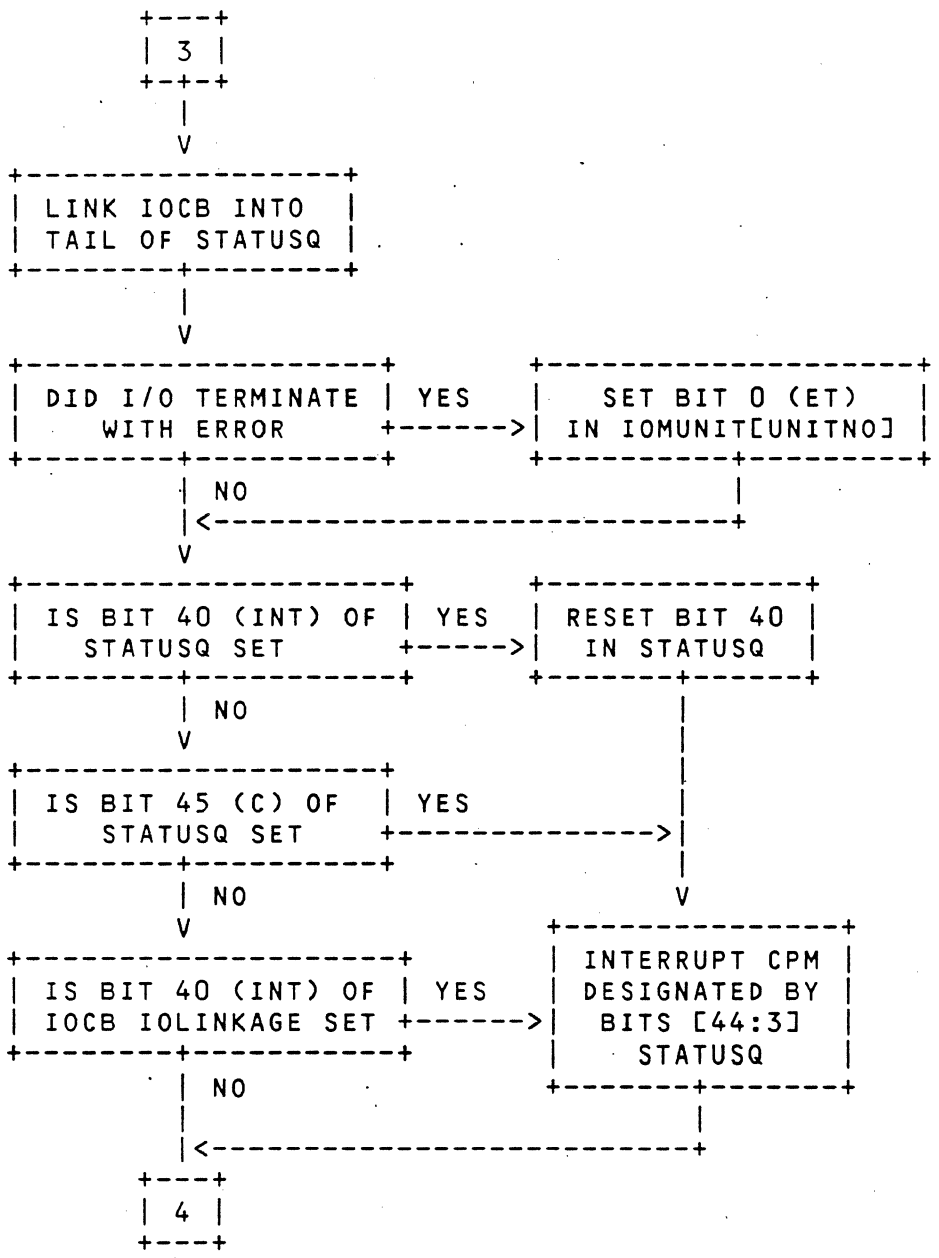


Figure 1-13. TERMINATE I/O (PAGE 2 OF 3)

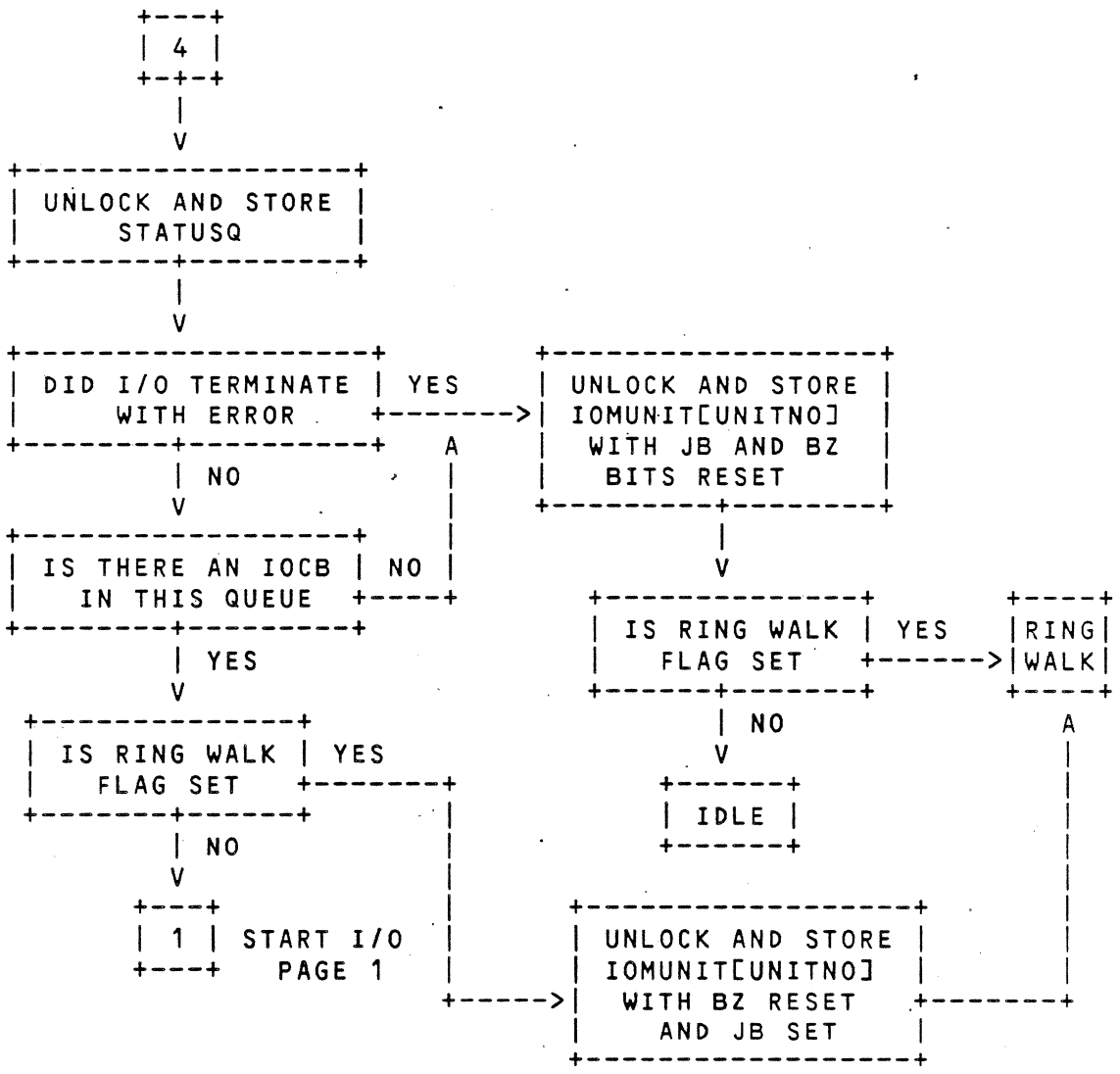


Figure 1-14. TERMINATE I/O (PAGE 3 OF 3)

B7800 TIGHTLY COUPLED SYSTEMS

The primary goal of tightly coupling is to achieve a multiprocessor system with more addressable memory than could be configured for a monolithic multiprocessor system. The 20 bit address field limits a monolithic system to one million words (6 MB) of addressable memory. A Tightly Coupled (TC) system, by making certain address ranges visible only to certain requestors, allows those ranges to be duplicated, and expands the total amount of memory accessible to the TC system to more than one million words. The B7800 implementation of Tightly Coupled Systems was accomplished under the restriction that it require absolutely no special hardware changes.

ARCHITECTURE

B7800 architecture dictated that a TC system be initialized (and run), as would any other multiprocessor system, from a single Halt/Load unit. A B7800 Tightly Coupled system has its low address memory visible to all on-line requestors; the MCM(s) comprising this shared memory constitute the SHARED or GLOBAL BOX. Above the highest SHARED mod is local memory: those MCMs which are visible only to specific requestors. A local BOX on a B7800 TC system consists of one or more CPMs, one or more IOMs and the high address MCM(s).

An important configuration restriction imposed on B7800 TC systems is that no unit may have in-use paths to more than one local box. Units may be exchanged across IOMs as long as the IOMs are in the same local box.

CONFIGURATION FILES

The configuration file is used to hold GROUP descriptions which define how the system is to be configured. The RECONFIGURE ODT command causes the MCP to assume a specified GROUP configuration.

The utility program, SYSTEM/CONFIGURATOR, is used to construct object configuration files (which can be used by the MCP) from symbolic configuration files. The internal file names used by CONFIGURATOR are SOURCE for the input file and OBJECT for the output file (defaults: input TITLE SYMBOL/CONFIGURATION, output TITLE SYSTEM/CONFIGURATION).

Input to SYSTEM/CONFIGURATOR consists of one or more GROUP descriptions, each with a unique GROUP name and its associated mainframe and peripheral specifications. The output file from SYSTEM/CONFIGURATOR is designated as the configuration file with the CF ODT command. The RECONFIGURE GROUP AS <GROUP Name> ODT command causes the configuration found to be displayed; an operator OK causes the MCP tables on the H/L

unit to be updated. The system will deadstop to allow the operator to make whatever changes are necessary to mainframe switch settings for the new configuration. When the system is loaded it will result in initialization based on the new configuration.

SUBSYSTEM DEFINITIONS

A TC system contains one GLOBAL memory subsystem and a local memory subsystem for each box. The local memory subsystems are identified by their BOX numbers and the GLOBAL memory is identified by the letter G or GLOBAL.

Subsystems may be given names in the form of alphanumeric identifiers. Subsystem identifiers are defined by the operator through the ODT command MS (Make Subsystem). Subsystems may be defined including any one or more memory subsystems in the system.

The user specifies a subsystem by using the task attribute SUBSYSTEM which may assigned a value only when the task is inactive. The SUBSYSTEM task attribute can be used in WFL, CANDE, programming languages and on Job Queues. The SUBSYSTEM attribute is treated as a request to guide placement of a task in memory.

B7900 HARDWARE OVERVIEW

This sub-section provides an overview of B7900 hardware and software. The information has been extracted from the 34 B7900 D-notes. In many cases the information has been copied from these D-notes. It is provided in this document for completeness.

INTRODUCTION

A B7900 system is comprised of several different modules. These include Central Processing Modules (CPM), Memory Subsystem Modules (MSM), I/O Subsystem Modules (IOSM), I/O Expansion Cabinets, a System Control Cabinet (SCC) and the System Console.

The MSM allows the B7900 to address more memory than any previous large system and allows simultaneous read/write operations.

The B7900 CPM is a pipelined processor with multiple (data and code) high speed cache (associative) memories.

The IOSM is comprised of two separate processing modules: the Auxiliary Processor (AP) and the Host Data Unit (HDU). The AP is a small, strictly serial processor which functions as a maintenance processor or in the B7900 system as an auxiliary processor.

The HDU manages all I/O operations on the system. The I/O subsystem it supports is the DLP-based Input/Output subsystem used by all of the 900 series systems. The IOSM also contains four base modules which contain the Data Link Processors which control the peripherals. The I/O Expansion Cabinet is used if additional base modules are required to configure the I/O subsystem. It contains up to six base modules.

The SCC houses three components: the master clock, the Maintenance Exchange (MEX), and the HDU/MSM profile card test station. The master clock provides clock signals to each of the boxes in the system. The MEX provides connection between the components which may act as "maintenance processors" and the maintenance interface in each box in the system.

The System Console provides table space for the system ODTs. For system initialization and maintenance, a modified MTS2 series terminal (MMTS2) is used. The MMTS2 acts as a maintenance processor, using two ICMD drives packaged in the leg of the maintenance console for program and file storage. The MMTS2 has access to the MEX for system initialization and

maintenance, and to the I/O Testbus for off-line DLP testing.

In the text that follows, these new features are explained in terms of their impact on the hardware and software that comprise the B7900 system. This explanation is by no means comprehensive.

CENTRAL PROCESSING MODULE (CPM)

The B7900 CPM is a powerful pipelined processor with high speed code and data caches. It is object-code compatible with all previous B5000, B6000 and B7000 machines except in two respects: it does not support the six bit character data type (BCL) and, like the B7800, it does not support vector-mode.

In addition to these changes, the basic design of the B7900 includes many improvements over the B7800. The two most noticeable of these are the more accurate arithmetic operations and more efficient branch handling.

The B7900 CPM performs arithmetic operations more accurately than any previous machine (except the B5900). This increased accuracy should cause numerical algorithms to converge slightly faster and may cause minor differences in arithmetic results as compared to previous machines.

As the B7900 CPM is a pipelined processor, it has the traditional difficulty handling conditional branches within a program. It must always choose a particular direction for a branch and prepare to execute the proper code for that direction. For example, the B7800 always assumes that the condition of an IF statement in FORTRAN or ALGOL will be true and fills its pipeline with the code that would be executed based on this assumption. Performance of the pipeline is penalized when branches do not go according to their assumed direction. The B7900 mitigates this penalty by remembering the direction which the branch last took, and assuming the same branch will be taken the next time.

The B7900 CPM has an increased memory addressing capability over the B7800 CPM, which is accomplished by memory environment registers. Because of the 20-bit address field, the CPM still only addresses one million words at any one time, but it can address all of memory by changing its environment registers to access different million-word environments. Extended Memory is discussed in the Memory Management sub-section.

The CPM consists of the following hardware modules: Program Control Unit, Data Reference Unit, Execution Unit, Store Queue, Memory Access Unit, and Card Test Station.

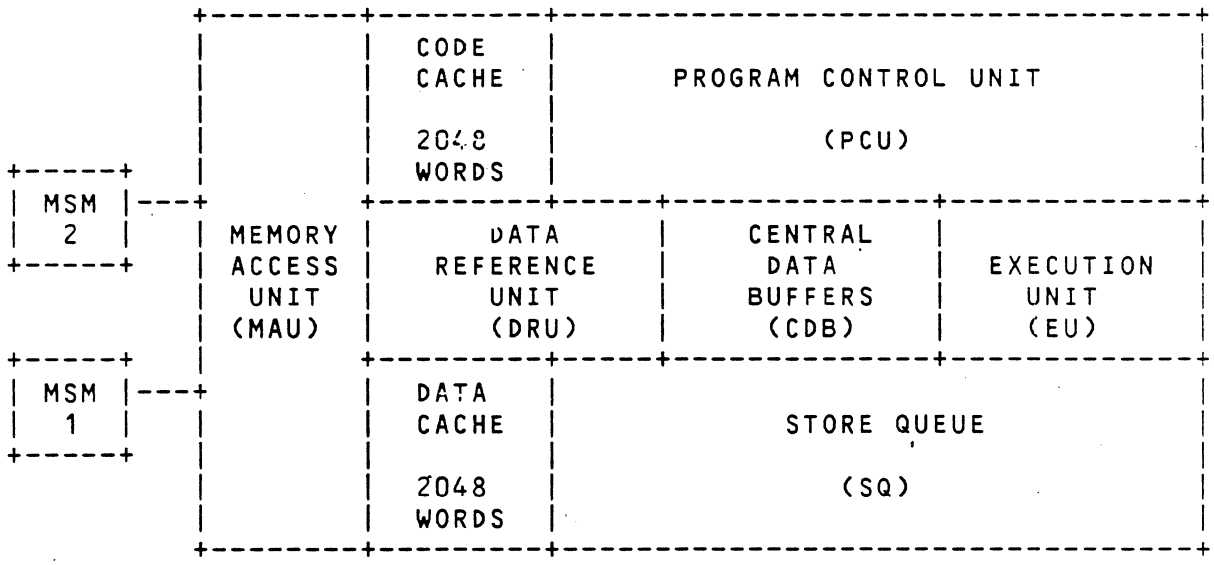


Figure 1-15. B7900 Central Processor Module (CPM)

The Program Control Unit (PCU) extracts code from the code stream and generates instructions for the Execution Unit and Data Reference Unit. These instructions are placed in queues along with a job number and addresses in the Central Data Buffer where input data can be found and where the result is to be stored. The PCU is the key unit for the pipeline by causing data to be ready for the Execution Unit ahead of time. Operator concatenation is also performed by the PCU.

The Data Reference Unit (DRU), upon command from the PCU, fetches data either from associative memory or main memory. Extensive use of pipelining allows one-clock references to associative memory.

The Execution Unit (EU) performs all arithmetic and logical operations using data in the Central Data Buffer. The result is either placed in the Central Data Buffer or may be sent to associative memory and the store queue.

The Store Queue (SQ) is used to reduce traffic to main memory by buffering data. Repeated stores to the same address will be performed as one store to main memory. Also, adjacent stores will be grouped into one multi-word store by the store queue.

The Memory Access Unit (MAU) provides the interface to main memory. The MAU can interface to two MSMs. On memory references the CPM Environment Register along with the 20-bit address are sent to the MSM.

The Card Test Station is used to run maintenance tests on CPM-style cards.

MEMORY SUBSYSTEM MODULE (MSM)

The B7900 includes one or two Memory Subsystem Modules (MSMs) totaling up to 96 megabytes. A MSM has a bandwidth of 72 MB/SEC. Each MSM contains from one to four Memory Storage Units (MSUs) and a Memory Control (MC).

The Memory Storage Unit (MSU) contains one or two sub-modules, each consisting of 6 megabytes. The memory is eight-word phased (transmission of words at clock rate) within each sub-module. This means the system can run with half an MSU or one sub-module without losing the speed of the eight-word phasing.

The Memory Control consists of the following major modules: Requestor Interface Adapter (RIA), Memory Interface Adapter (MIA), Priority Resolution Module, Error Module, Requestor Interrupt Module and Maintenance Module.

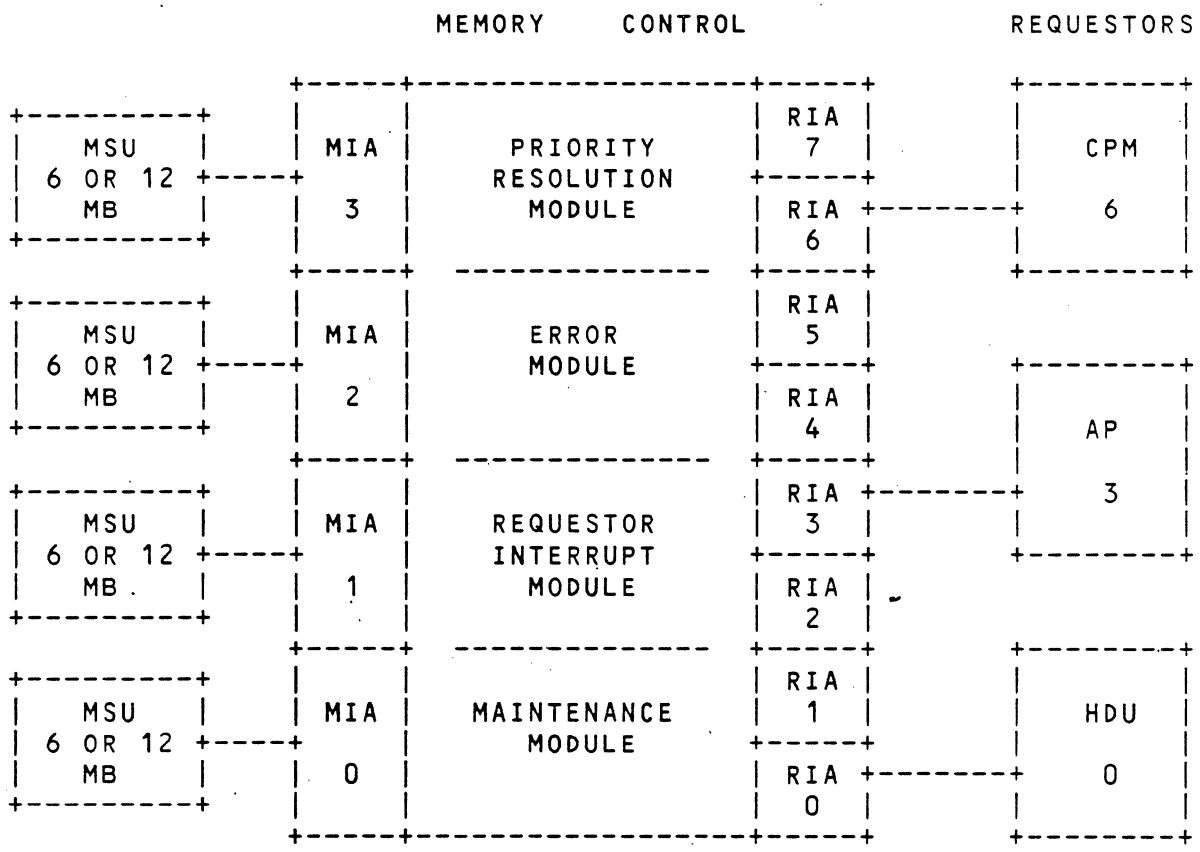


Figure 1-16. B7900 Memory Subsystem Module (MSM)

Each requestor interfaces to memory through its own Requestor Interface Adapter (RIA). For any memory operation, a control word and up to eight data words are sent by the requestor to its RIA. The RIA checks these words for errors, reports any errors to the requestor, and stores them in a buffer memory until the priority resolver commands their transmission to the Memory Interface Adapter (MIA).

Each RIA contains an address translation table which determines the validity of the address for this MSM and where the address resides physically in storage. That is, the address translation table provides a mapping from environment number and address to physical location in memory. Each address translation table contains 1024 eight bit words. Each entry provides information for a page (128K words) of memory. The table entry contains a MSU number (2 bits), MSU relative address (4 bits), validity bit and parity bit.

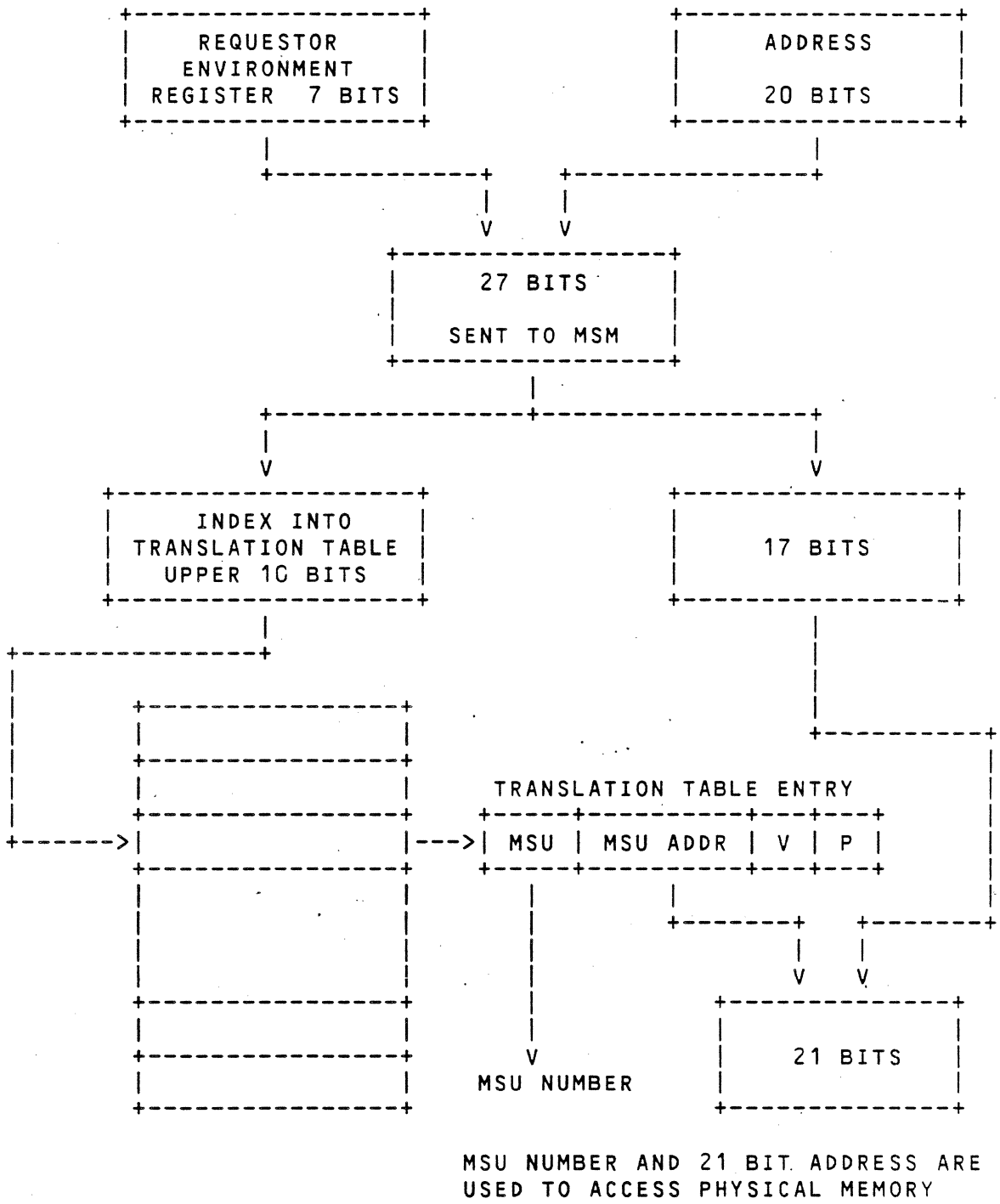


Figure 1-17. MSM Translation Table

The Memory Interface Adapter (MIA) controls the flow of information between the memory control internal bus structure and the Memory Storage Unit (MSU). A memory control contains from one to four MIAs with the capability of all MIAs making simultaneous requests to the MSU to which they are connected.

The order in which system requests are handled by the MSM is determined by the Priority Resolver Module. Requests are processed by a modified First In First Out (FIFO) algorithm. Snapshots of memory requests are recorded on a basic time interval as they arrive. The snapshots of requests are processed in the order they are taken.

Within a snapshot, requests will be serviced in the following sequence: read operations, write operations, special operations (except for write translation table), write translation table. The higher number requestor will be given the higher priority, if there are duplicate requests for a particular type, although this should have no effect on system performance.

By means of a modified Hamming Code, the Error Module can detect and correct single bit errors and detect two-bit errors. Any errors detected during MSU operations or internal memory control operations are reported to the RIA. The Error Module will report the control word or the fail word or both if requested by the requestor.

The requestor interrupt module is responsible for the correct routing of interrupts and for interrupting the destination requestor via unique interrupt lines on receipt of an incoming interrupt. There are two types of interrupt buffers maintained by the MSM for each requestor: hardware and software. These buffers are used to accumulate interrupts and can be read by the requestor via a system operation. The software interrupt buffer contains a bit mask of requestors from which interrupts have been received. The hardware interrupt buffer is an eight-bit mask which designates the type of interrupt.

The maintenance module performs all maintenance type functions for the MSM. In system mode, the maintenance module handles reading and writing the fields in the MSM BOXID. All other maintenance operations are done through the maintenance interface.

AUXILIARY PROCESSOR (AP/AMP)

The Auxiliary Processor/Auxiliary Maintenance Processor (AP/AMP) is based on the B5900 Entry Level System and is compatible with the B7900 CPM. The AP/AMP has two distinct modes of operation: AP and AMP. To support these two

different modes two unique versions of micro-code, which communicate with different sets of hardware, are required.

In Auxiliary Processor (AP) mode, the AP/AMP acts as a co-processor to the B7900 CPM with the purpose of handling I/O finishes when the CPM is busy. The system has the capability of running with no CPMs (AP only mode). The AP will perform all the functions a CPM would do in this case. When the last CPM is removed from the system, the MCP notifies the operator (with an RSVP) that no CPMs remain. The operator must OK this waiting entry in order to allow jobs to run on the AP.

In Auxiliary Maintenance Processor (AMP) mode, the AP/AMP utilizes its own local memory and HDP hardware section of the processor to exist as a standalone system. The primary function of the AMP is to perform maintenance on the B7900 system. The AMP operationally is the same as the B5900.

The AP/AMP can be divided into five major units: Processor, Local Memory (LM), Cache and System Memory Controller (APMC), Interrupt and Maintenance (APIM) and the Host Dependent Port (HDP).

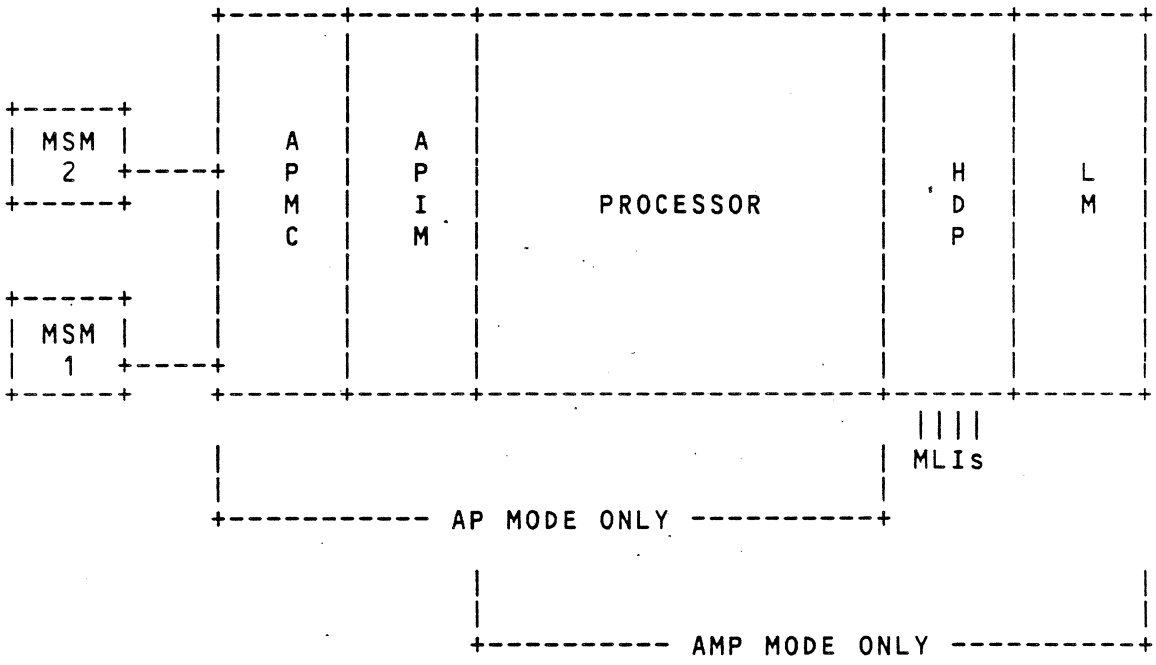


Figure 1-18. Auxiliary Processor (AP/AMP)

The processor module performes all arithmetic and logical operations for the AP/AMP.

When the AP/AMP is running in AMP mode, the HDP section becomes active. The HDP provides the I/O subsystem interface

for the AMP.

When the AP/AMP is running in AMP mode, the local memory becomes active and the B7900 MSM memory is no longer used. The local memory has a capacity of 512K words. Error correction and logging are features provided by the LM.

When the AP/AMP is running in AP mode, the AP interfaces to the B7900 MSM memory by means of the APMC (Cache and System Memory Controller) module. The APMC can interface with two MSMs. The cache memory consists of 128 words of code and 128 words of data. The cache hardware performs automatic prefetching of code. Any word in the cache can be accessed in 1 clock and the cache can be purged in 2 clocks.

The AP/AMP Interrupt and Maintenance Module (APIM) interfaces to the MSM system interrupt bus and Maintenance Exchange (MEX), contains AP box identification information, and provides support for the AP Memory Control/Cache (APMC).

HOST DATA UNIT (HDU)

The HDU contains a Memory Bus Control (MBC) which handles the HDU's interface to the MSM, the Queue Manager (QM) which handles the I/O queue structures and flow control, and three Host Dependent Ports (HDPs) which connect to the I/O subsystem. Each HDP is capable of operating at a burst rate of 8 Megabytes/Second for a total of 24 Megabytes/Second per HDU.

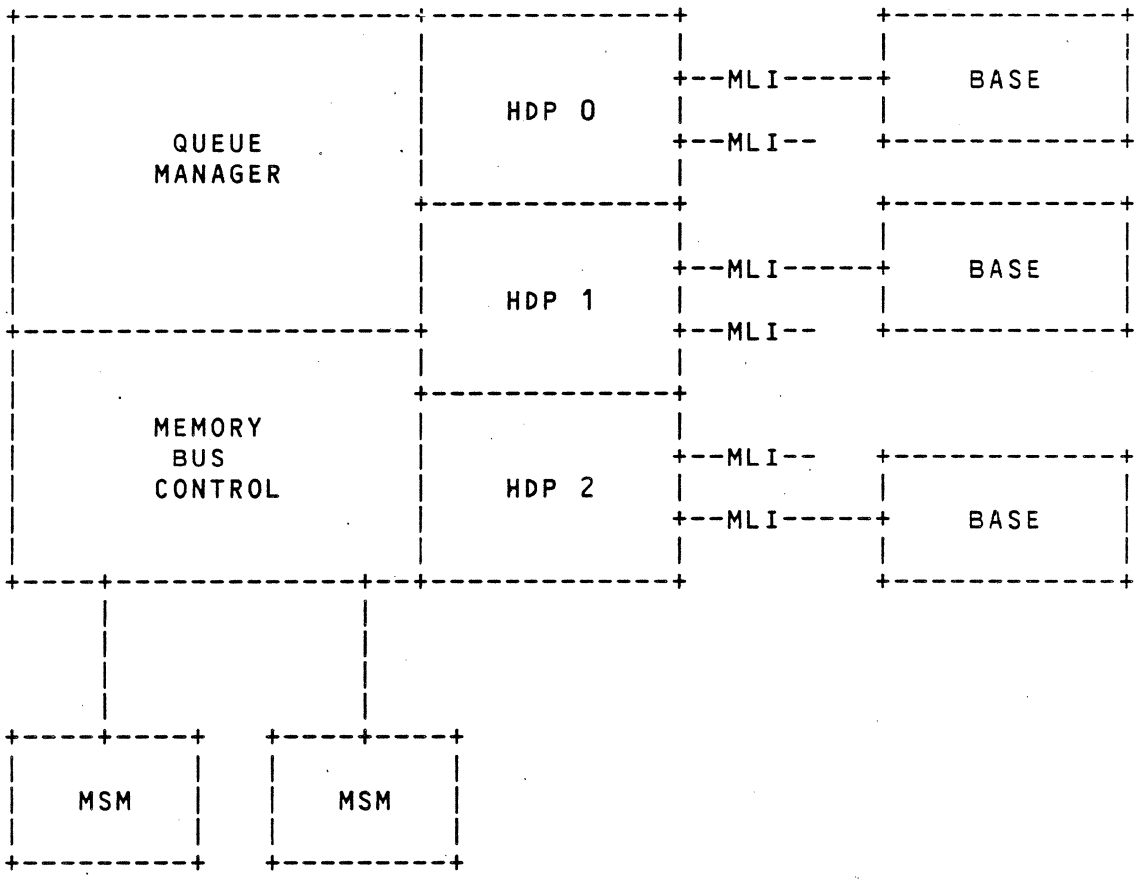


Figure 1-19. Host Data Unit (HDU)

The Queue Manager (QM) is responsible for managing the I/O queue structures and the scheduling of jobs for the HDPs. It has a 4K word local memory which is used to selectively store copies of the IOCBs that have been issued to the I/O subsystem. 256 words of this local memory are used to contain the code for the MCP Boot and the Minimal Configuration information needed to initialize the system. It is called the HDU's Halt Load RAM.

The Memory Bus Control (MBC) provides the HDU's interface to the memory subsystem. It can connect to up to two MSMs. The MBC handles up to eight-word memory transfers and uses the same memory addressing mechanism (Environment Register plus 20-bit address) as the CPM. It also handles the requestor to requestor interrupts which use the memory buses.

The Host Dependent Port (HDP) provides the HDU's interface to the I/O subsystem using the Message Level Interface (MLI). Each HDP controls two MLI ports, each of which connects to one

MLI cable. The HDPs control the MLI and are responsible for all data transfer for the system.

DLP-BASED I/O SUBSYSTEM COMPONENTS

Each HDP in the HDU contains two MLI ports which are connected to I/O bases via an MLI. An I/O Base is a rack which contains a Base Control Card (BCC), Distribution Cards (DCs), a Maintenance Card (MC), Data Link Processors (DLPs), and, optionally, a Path Selection Module (PSM), and/or Line Expansion Module (LEM).

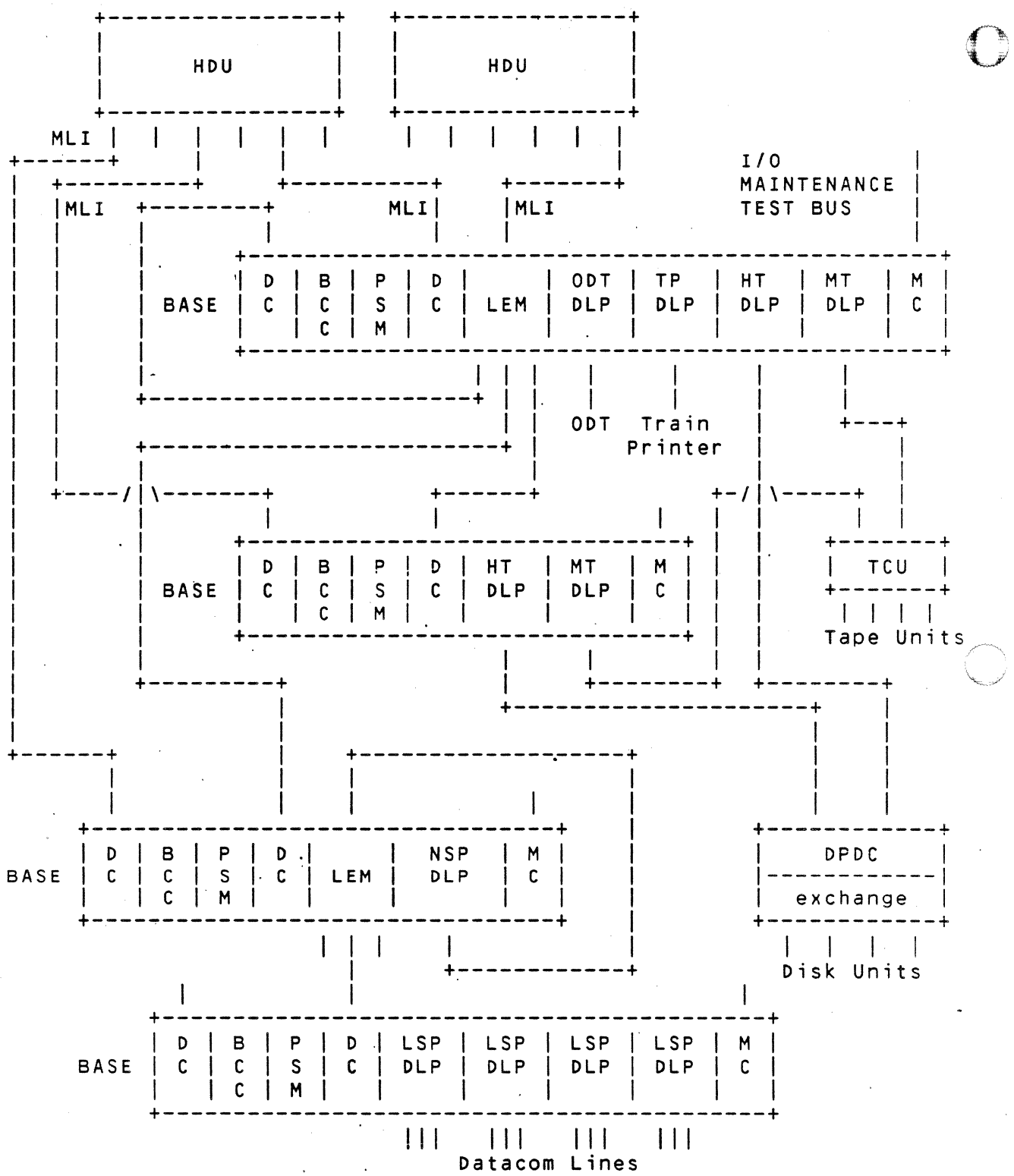


Figure 1-20. UIO Subsystem Components

The Maintenance Card (MC) provides maintenance functions for the base. It connects to the I/O Maintenance Test Bus which is used to perform off-line maintenance.

A Distribution Card (DC) provides the interface between the base module and the host. One MLI cable from a host's MLI Port connects to a DC in the Base Module. Through the DC, a host can communicate with the BCC and the DLPs in the base. There can be only six DCs in a base although the "addressing" allows for eight. DCs are "addressed" by jumpering an ID from 0 to 7.

A Path Selection Module (PSM) coordinates the activities of multiple DCs in a base, and is required in a base which has more than one DC. Several hosts can be actively communicating with DLPs in a base, but only one DC can access the BCC or have access to the base backplane at any time. The PSM resolves these conflicts.

The PSM also uses the masks in the BCC in order to clear only the DLPs assigned to the host issuing a Master Clear by issuing a Selective Clear or Selective Master Clear (see below) to each assigned DLP.

The PSM contains eight field-installed jumpers (one for each possible DLP in the Base) which, when set, will cause a Selective Master Clear (instead of a Selective Clear) to be issued to the corresponding DLP when the host system issues a Master Clear.

The Base Control Card (BCC) provides identification and access control for a Base Module. The BCC contains two types of masks which are used to control access into a base and its DLPs. These are the DC Enable Mask and the DLP Enable Masks.

The eight-bit DC Enable Mask indicates which DCs are allowed access to the base. A locked-out DC will not respond to the host or execute a host-generated Master Clear.

The BCC also has eight 16-bit DLP Enable Masks, one for each possible DC in the base. A DC's DLP Enable Mask indicates to the PSM which DLPs this DC is allowed to access. The PSM also uses these masks in order to determine which DLPs will be cleared when a MLI Master Clear is received via a DC. Only those DLPs enabled to that DC will be cleared.

Along with the eight DLP Enable Masks, the BCC contains an Acquire Enable Bit for each possible DC in the base. This bit indicates whether or not another DC may "steal" DLPs from this DC.

The BCC also has a 32-bit Maintenance Mask that indicates which devices in the base (DLPs, DCs, BCC, PSM, LEM) are enabled on the I/O Maintenance Test Bus. Devices are

identified in the mask by their Maintenance Addresses (or Unit Numbers).

The DC Enable Mask and the DLP Enable Masks can be cleared only by one of the following manual operations:

1. Powering up the Base.

2. Pushing the RCC Clear Switch on the BCC.

3. Pushing the Base Clear Switch on the MC.

When the BCC is cleared, the DC Enable mask is set so all DCs are enabled in the Base, the DLP Enable masks are set so all DLPs are disabled from all DCs, all Acquire Enable bits are set to disable the "steal" function and Maintenance is enabled to all devices in the Base.

The DC Enable Mask can be set only via a DC (by software) if it is enabled in the DC Enable Mask. A DC can enable a DLP in its DLP Enable Mask only if the DLP is not already enabled in another DC's DLP Enable Mask. The exception to this rule is the ability to "steal" a DLP. In order for a DC (DC 1) to steal DLPs from another DC (DC 2), DC 2's host must have previously set DC 2's Acquire Enable bit. When DC 1's host issues a variant of the LOAD DLP MASK command, the specified DLPs will be disabled from DC 2's DLP Enable mask and enabled in DC 1's DLP Enable Mask. A DC may also share DLPs with other DCs and transfer DLPs to another DC.

A Line Expansion Module (LEM) enables a host to be connected to more bases by expanding one MLI to up to seven MLIs. It resides in a base.

A LEM consists of three types of cards. The type "A" card contains the basic LEM logic, two MLI ports (ports 6 and 7) and the Maintenance Address jumpers.

The 1x2 LEM ID, or type "C" card, provides two MLI ports (ports 1 and 2) and LEM identification. The 1x3 LEM Exchange, or type "B" card, provides three MLI Ports (ports 3, 4 and 5).

A Data Link Processor (DLP) provides the interface between hosts and peripheral devices. The operation of a DLP is device-dependent, and there are different types of DLPs to control different types of devices. All DLPs follow the MLI protocols to communicate with a host, perform a set of standard DLP operations, and return certain standard results. Included in these standard DLP operations is the TEST IDENTIFY operation which returns the DLP type and the DLP's strapped ID.

The DLPs that are currently supported on the B7900 are:

DLP	Peripheral
CR1	Card Reader
HC2	ISC Host Control
HT1	(Host Transfer) 206,207,659,677 Disk Pack
MT1	Mag Tape: PE
MT2	Mag Tape: PE/GCR
MT3	Mag Tape: NRZ
ODT1	Operator Display Terminal
LSP1	Datacom Line Support: Sub-broadband
TP2	Buffered Printer: 1200/2000 LPM
NSP3	Datacom NSP: Blocked

There can currently be up to eight DLPs in a base although there are 16 "addresses" reserved. A DLP's "address" is determined from jumpers on the DLP's logic cards and is also its Maintenance Address.

DLP address assignment affects hardware priority algorithms. For proper operation of the system, these algorithms must be understood and the DLP addresses assigned accordingly.

Within each Base, concurrent requests by multiple DLPs are arbitrated by the Base Control Card based on the Base Module Priority of the DLPs. The Base Module Priority of a DLP is determined directly by its address; the DLP with address 7 has a Base Module Priority of 7, which is the highest possible priority. When presented with multiple DLP requests, the Base Control Card selects the DLP with the highest Base Module Priority.

DEVICE NAMING

The B7900 will have its I/O configuration described in a Peripheral Configuration Diagram (PCD). The PCD is the B7900 equivalent of the B7700/B7800 Unitcards and contains descriptions of all devices and connections in the peripheral subsystem. In the PCD all devices (including units, DLPs, LEMs, Bases and MLI Ports) are assigned arbitrary numbers, called Device Numbers, which will be used when referring to a device. The PCD also contains the information necessary to determine the paths to the device to be used in performing I/O operations.

The B7900 supports larger device numbers than previous systems, which supported eight-bit device numbers. Device numbers for the B7900 are currently 15-bit numbers (or between 1 and 32767). Although devices may be numbered between 1 and 32767, only 255 peripherals, NSPs and LSPs may be in a B7900

partition at one time. The B6800, B5900/B6900 and B7700/B7800 systems are still restricted to eight-bit device numbers (between 1 and 255).

The device number for an NSP or LSP must be an integer from 1 to 255. This device number must match the NDLII program. An NSP can control up to 8 LSPs at any time.

The device number for an HC must be an integer from 1 to 255 if BNA is to be used.

The B7900 MCP will use the PCD to determine the paths to a device and will attempt to verify the information in the PCD before using a device.

AMP DEVICE NAMING

The B7900 AMP, which is a modified B5900, uses the B5900 device naming rules. B5900 systems use the strapped ids in the I/O subsystem components in order to identify or name the peripheral units controlled by them. Currently the MCP only allows for eight-bit device numbers for the B5900. If a DLP can have only one unit attached to it, the lower eight bits of that unit's number are the same as the DLP's strapped ID. If there can be more than one unit controlled by a DLP, the unit's number is the DLP's strapped ID plus the unit's address relative to the DLP.

Peripherals not qualified on the B5900 are not supported on the AMP.

I/O RECONFIGURATION

Online I/O reconfiguration on the B7900 has similar capabilities to the B7800. Many of the functions are identical, in syntax and semantics. Because of the difference in the hardware components, though, there are some differences in syntax and functionality, and there are several new and unique functions.

B7900 peripheral reconfiguration (FREE, ACQUIRE, RY, SV, UR, UR-, MOVE, REPLACE and CL) is identical to B7800 peripheral reconfiguration with two exceptions. The first is that the RY command with path selection to append devices and the SV (-) command to remove them are not supported. These functions can be accomplished by loading a new Peripheral Configuration Diagram (PCD). The second is that the syntax of the LH (Load Host) command (to load a controlware file to a disk pack drive controller) is slightly different. The DLP device number is used to specify the path instead of CH <#> IOM <#>.

There are two B7900 I/O reconfiguration capabilities which are functionally similar to B7800 capabilities. FREE/ACQUIRE IOP (IOP is used to refer to an HDU in the B7900 I/O ODT commands) corresponds to FREE/ACQUIRE IOM. UR DLP <dlp device number> is similar to UR CH <#> IOM <#>.

The unique B7900 I/O reconfiguration capabilities include FREE/ACQUIRE DLP, UR/UR- MLI, UR/UR- BASE, UR/UR- BASE MAINT and UR/UR- DLP MAINT.

DLPs are not shareable between partitions and, therefore, must only be ACQUIREd by one partition at any time (FREE/ACQUIRE DLP). DLPs may be reserved in order to perform on-line maintenance or to suspend operations to the DLP via UR/UR- DLP. Off-line maintenance via the I/O Maintenance Test Bus can be performed using UR/UR- DLP MAINT.

Bases are shared by all B7900 partitions to which they are connected. Bases may be reserved in order to suspend operations through the Base or reserved for off-line maintenance.

MLIs are implicitly ACQUIREd with the HDUs to which they are connected. They may be reserved in order to suspend operations through a particular MLI.

I/O UNIT/DLP/MLI LOAD BALANCING

Each Base typically has multiple Distribution Cards and, thus, there are typically multiple paths into each Base. Each of these paths is connected directly to an MLI or fans out from an MLI via a LEM. Between a running partition and a particular Base, there is at any given time a set of viable MLIs. This set of MLIs is relatively static; it can change only when HDUs are FREEd/ACQUIREd or MLIs are URed/UR-ed/BROKEN. The MCP automatically attempts to evenly distribute I/O traffic over all viable MLIs. The balancing algorithm is always active and does not require operator action.

Pack and tape units (which can be connected to an exchange) may be accessible by more than one DLP. (Note: each DLP or peripheral should be ACQUIREd in only one partition at a time.) Between a running partition and a particular string of exchanged units, there is at any given time a set of viable DLPs. This set of DLPs is relatively static; it can change only when HDUs, MLIs, Bases, or DLPs are reconfigured or marked BROKEN. The MCP automatically attempts to evenly distribute I/O traffic over all viable DLPs. The balancing algorithm is always active and does not require operator action.

System Option 39 (PATHBALANCING) has no effect on B7000

Systems, including B7900 Systems.

I/O TIME

Reported "I/O Time" for tasks using moving-head media (e.g., packs) will generally be greater on HDU Systems (e.g., B7900) than on IOM Systems. This discrepancy results from the different handling of seek operations by the two I/O subsystems and does not indicate that the actual I/O operations are slower on the B7900.

On IOM Systems, I/O Time on moving-head media is the sum of rotational latency and data transfer time. On HDU Systems, I/O Time on moving-head media is the sum of seek time, rotational latency, and data transfer time.

B7900 I/O FLOW AND I/O QUEUEING STRUCTURES

The following is a brief discussion of I/O flow on the B7900. The I/O flow involves many different queueing structures. The queues manipulate the flow of Hardware Control Blocks (HCBs). The purpose of each queue will be discussed as well as its structure. All the queues reside in main memory except for the HDP queues which reside in the local memory of the HDU's Queue Manager. Two of the queues, the Home Address Queue (HAQ) and the Result Queue (RQ), form the primary interface between the MCP and the HDU.

Unlike the B7800, it should be noted the path to get to any peripheral device on the system is solely determined by the MCP. The information about the path to be used is placed in the HCB before the HCB is queued into the Home Address Queue. The HDU cannot change this predetermined path.

HARDWARE CONTROL BLOCK (HCB)

The first structure to be discussed is the Hardware Control Block, hereafter referred to as HCB. An IOCB is generated for each I/O operation on the system. A HCB will be generated and linked to the IOCB. The IOCB usually resides in local memory where as all HCBs reside in the shared memory component of the system. The HCB is used to pass information to the hardware so the I/O can be completed. HCBs are 24 words in length of which the first 16 words are used by the hardware. The other eight words contain information used by the software, including a link back to the original IOCB.

The MCP manages a pool of available HCBs. There is a global pool as well as a pool for each processor. Because separate pools are set up for each processor no locking is required to get or release an HCB. There is also a mechanism where a

processor can return HCBs from its pool back to the global pool. There is a limit set on how big each processors pool can get.

The HCB pool is managed by the procedures ALLOCATEHCB and DEALLOCATEHCB. When ALLOCATEHCB cannot find an available slot in the pool for an HCB it will make a call on GETAREA to allocate another 125 word area to add to the pool. The procedure DEALLOCATEHCB is called when an HCB is being returned to the pool. Once an area is obtained by GETAREA for the pool it is never released. This was done for performance reasons because the overhead to manage these areas could be great. Also, there is no attempt to always keep available areas in the pool, they will be allocated as they are needed.

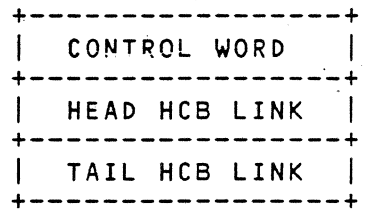
The allocation of HCBs for normal I/O operations occurs in the procedure called INITIATEIOOPERATION. ALLOCATEHCB is called to find an area for the HCB from the pool of HCBs. Once the HCB is acquired, INITIATEIOOPERATION calls the procedure SETUPHCBFROMIOCB to fill in the proper information in the HCB. The HCB is then inserted in the Home Address Queue (HAQ). If the HAQ is empty when the HCB is inserted the HDU is interrupted. The HDU will delink all HCBs from the HAQ queue until the queue becomes empty. The hardware has now taken over the handling of the I/O. The Queue Manager of the HDU manages the I/O queues from this point.

When the HDU completes the I/O process the HCB is returned through the Result Queue (RQ). The processor is interrupted to delink the HCBs from the Result Queue and process the IOFINISH. For a normal I/O, the procedure TERMINATEIOOPERATION will pass the information on the I/O completion back to the original IOCB and return the HCB to the pool.

HOME ADDRESS QUEUE (HAQ)

The Home Address Queue (HAQ) is a linked list of HCBs. There is one HAQ for each HDU on the system. The HCBs are placed in the queue by the MCP and are delinked by the HDU.

The HCBs in the queue are linked to each other via a link word in the HCB itself. The queue is composed of a control word, a head word and a tail word. In the control word bit [0:1] is called the lock bit, this bit determines control of the queue by the MCP or HDU. If the HAQ is empty and an HCB is linked into it the HDU will be interrupted. The HDU will delink HCBs from the HAQ and insert them into the proper Command Queue (CQ).



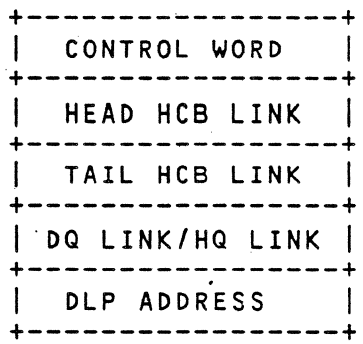
COMMAND QUEUE (CQ)

There is one CQ for each peripheral device on the system. When the HDU delinks the HCBs from the HAQ it places the HCBs in their appropriate CQs.

The CQ is described by a Command Queue Header. This header contains five words. The first word is a control word, this word contains an active count and an active limit field as well as other control information. The active count field indicates the number of I/Os currently in progress. The active limit field indicates the maximum number of I/Os the device can process simultaneously. For most devices the active limit will be one.

The other words in the header include a head word, a tail word, a DLP Queue Head Pointer or an HDP Queue Pointer and a DLP Address Word. The head and tail words are used to link the HCBs into the queue. The DLP Queue Head Pointer points to the DLP Queue (DQ) this device is associated with. If this is a non-exchanged device there will be no DQ, the HDP Queue Pointer would be found in the header. The DLP Address Word contains the address of the DLP servicing the device.

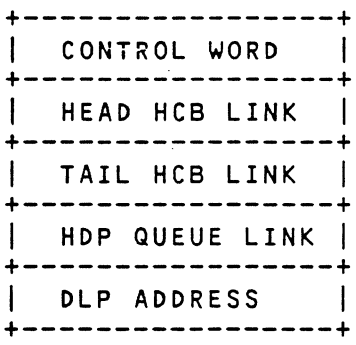
When HCBs are queued in the CQ and the active count is less than the active limit, the first HCB in the queue is linked into the appropriate DQ and delinked from the CQ. If no DQ is present the CQ is linked into the appropriate HDP Queue (HQ). The active count is incremented.



DLP Queue (DQ)

There is a DLP Queue (DQ) for each DLP which services multiple devices. A DQ is used to enqueue HCBs that have been initiated, but have not actually been sent to the DLP. The DQ is described by a DLP Queue Header. This header consists of five words. The first word is a control word which has the same format as the control word for a CQ. The active limit for the DQ would be the sum of the active limits of the CQs being serviced by this DLP. The header also has head and tail link words, an HDP Queue (HQ) link word and a DLP address word.

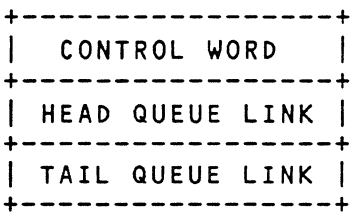
When the DQ has HCBs queued the DQ is linked into the HDP queue (HQ) for the HDP that is currently servicing the DLP.



HDP QUEUE (HQ)

There is a HDP Queue (HQ) for each HDP configured with the HDU. The HQs are contained in the local memory of the Queue Manager and are therefore never available to the MCP. Whenever a HQ becomes nonempty the Queue Manager will interrupt the HDP. The HDP will then process the I/Os in the HQ until the queue becomes empty.

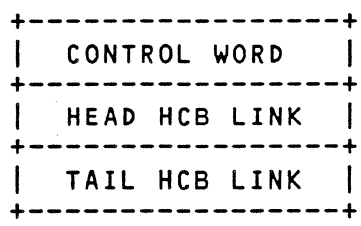
A HDP Queue is queue of DQs and/or CQs. The queue consists of three words, a control word, a head queue link and a tail queue link.



RESULT QUEUE (RQ)

When the I/O for an associated HCB is completed, whether it be a successful completion or not, the HCB is then linked into the Result Queue (RQ). Each HDU has its own RQ. The processor will be interrupted to handle the IOFINISH operation for the completed I/Os.

The RQ consists of three words. A control word, a head HCB link and a tail HCB link. Bit [0:1] of the control word is a lock bit. This bit determines who has access to the queue at any point in time, the MCP or HDU. The control word also contains a CPM number. This is the number of the CPM or AP that the HDU will interrupt to handle the IOFINISH operations.



ERROR COMMAND QUEUE (ECQ)

There is a special queue on the system called the Error Command Queue (ECQ). This is a special purpose queue that contains a pool of HCBs. HCBs are withdrawn from the queue by the HDU and used to report errors not associated with a particular device, for example HDU errors. The MCP is responsible for maintaining a reserve of HCBs in the queue for this purpose.

SUMMARY OF I/O FLOW

The following is a brief general recap of the flow of an I/O on the B7900.

1. When an I/O is to be initiated an IOCB is set up.

2. The MCP is passed the IOCB to be initiated. The MCP will generate an HCB and link it to the IOCB. This HCB is then linked into the Home Address Queue (HAQ).

3. When the HAQ becomes nonempty the HDU is interrupted. The Queue Manager in the HDU is then responsible for delinking the HCB and continuing the I/O process.

4. When the HCB is delinked from the HAQ it is linked into the appropriate Command Queue (CQ). There is a CQ for each device on the system.

5. When the device associated with this CQ can process the HCB, the HCB is delinked from the CQ and linked into the

DLP queue (DQ) for this CQ. The DQ is the queue for the DLP currently servicing the device. If the device is a non-exchanged device it will not have a DQ and the CQ will be linked directly into the HDP Queue (HQ).

6. If the DQ has HCBs queued it is linked into the HDP queue (HQ) associated with it. The HQ is read by the HDP.

7. Once linkage is made to the HQ the HDP will process the I/O. When the I/O is completed the HCB will be placed in the Result Queue (RQ).

8. The processor will be interrupted. The HCB will be delinked from the RQ and the IOFINISH operation will be completed.

For more information on the layout of an HCB or words associated with the various queues, refer to B7900 IOSM Technical Manual, Volume 1: Operation and Maintenance.

MAINTENANCE HARDWARE

The B7900 incorporates "soft" access to low-level machine state. Using this mechanism, initialization and maintenance software may interrogate and set flip-flops, registers and RAM storages in the various boxes in the system.

Within any system, every box is cabled to the System Control Cabinet (SCC) which houses the Maintenance Exchange (MEX). The MEX provides for connection from any of the "maintenance processors" in the system to each of the boxes in the system. The overall management of the maintenance bus is a "master-slave" relationship, where "maintenance processors" are masters and the various boxes they access are slaves. The MEX includes Fan In Adaptors (FIAs) and Fan Out Adaptors (FOAs). There is a FIA for each of the possible masters, and a FOA for each of the slaves in the system. Within each box (slave) there is "on-board" maintenance interface logic to allow for state access to that box. The overall arrangement of FIAs, MEX, FOAs and on-board maintenance logic is commonly called the "maintenance bus".

A MIP is a hardware component housed in the "leg" of a B7900 Console. It is cabled to a pair of ICMD (floppy) drives, also housed in the console leg. The MIP may also be cabled to the I/O Testbus. A MIP is driven by a Modified MTS2 terminal; only one MMTS2 may be cabled to any particular MIP, so the MMTS2/MIP will typically be paired and thought of as one unit. A MMTS2/MIP which is cabled to the MEX may act as a "maintenance processor" for the system.

In any system, every processor will be cabled as a master to the MEX. This allows any "running" B7900 partition to act as

a "maintenance processor" for hardware maintenance of boxes not in the partition. The maintenance software used in this mode is IDA.

In any system, one or more APs will be configurable as AMPs. Using its connection to the MEX, an AMP may act as "maintenance processor"; this is done when it is not possible to configure a B7900 partition (for example, all MSMs down). The maintenance software used in this mode is IDA.

Built into the CPM cabinet, and appearing over the maintenance bus as another "slave" station, is the CPM profile card test station. The software used for CPM card testing is IDA.

MAINFRAME MAINTENANCE

Mainframe maintenance for the B7900 is divided into two groups: the AP/AMP and the other mainframe boxes (MSM, HDU, CPM).

Maintenance for the AP/AMP is done using the MMTS2/MIP on the System Console. Two software packages are used: BEAM and APCON. BEAM provides for "low-level" static testing of the AP/AMP. The test patterns for these static tests are packaged on a set of ICMD diskettes, and the BEAM program uses the MIP to run the patterns against the AP/AMP hardware. APCON provides for dynamic tests. The dynamic tests are written in AP/AMP microcode, and execute on the AP/AMP itself, with APCON serving to load the tests from their ICMD diskette and to monitor their execution.

The CPM, MSM and HDU are maintained from a "running system", using the IDA software. IDA is an ALGOL program, supported by a set of maintenance libraries. It runs under MCP control, using normal system resources (such as CANDE). When enough hardware remains operational to configure and run a B7900 partition, IDA is executed on that partition. When it is not possible to configure a B7900 partition (for example, in a model B7900F with the MSM down), the AP/AMP is configured and run as an AMP, and IDA executes there. In either case, IDA uses the system processor's connection to the MEX to gain maintenance bus access to the box under test. Both dynamic and static tests are provided for each of the box types. IDA is also used to drive the two card test stations: the CPM profile test station, packaged in the CPM itself, and the HDU/MSM profile test station, packaged in the SCC.

I/O SUBSYSTEM MAINTENANCE

The B7900 I/O subsystem will be maintained either in an off-line mode or an on-line mode. Off-line maintenance allows for the execution of diagnostic tests on the I/O components in a Base. These tests are run from the MMTS2/MIP on the Maintenance Console, accessing state in the off-line device via the I/O Maintenance Test Bus. The test patterns are shipped on ICMD diskettes, along with the various I/O components.

On-line maintenance allows for confidence and maintenance testing of DLPs and the peripherals they control. These tests will be performed by running Peripheral Test Language (PTL) programs through Peripheral Test Driver (PTD) under MCP control.

PARTITIONING

In a B7900 system, "Partitioning" provides a means whereby a subset of the boxes in the system may be managed by a single MCP. Moreover, partitioning prevents any box not in the subset from in any way interfering with the successful execution of the subset. Such a subset is called a partition. This capability may be used in two ways:

> A broken box may be tested while the remainder of the boxes executes as a partition. In this scheme the partition is protected against interference from the broken box.
>
> A given system may be "split" into more than one partition, with each partition run independently. In this scheme each partition is protected against interference from actions in the other partitions.

Partitioning is established by software, but enforced by hardware. It involves the control of three interfaces: the system interface, the I/O subsystem interface, and the maintenance interface. Partition membership is marked in the Partition Identification Register (PID) in each box.

For example, in a Model H or K B7900 with the PAC kit, the system can be run as a single partition, including all boxes. A "broken" box may be removed from the partition, tested, and returned without jeopardizing the continued execution of the partition. Further, the system may be "split" into two partitions; each half would be run completely independently of the other.

The software involved in making partitions are SYCON and the MCP. SYCON will initially establish a partition upon

instructions from the user. This includes setting the controls for the system and maintenance interface, and the PIDs in all boxes in the partition. SYCON will then start the MCP running. SYCON is further discussed in the section on System Initialization.

The MCP will, as part of its initialization, establish control over the maintenance interface, thus disabling the maintenance bus and excluding the "maintenance processors" from further access over the bus.

Boxes may be moved into a running partition only by the MCP executing in that partition, using the MCP's ACQUIRE command. During this acquisition, the system interface controls in the boxes currently in the partition will be updated to acknowledge the existence of the incoming box, and the system interface controls in the incoming box will be updated to reflect the partition's membership. The PID in the incoming box will be set to match the partition number. Similarly, a box may be removed from a running partition only by the MCP executing in that partition, using the MCP's FREE command. During that process all boxes remaining in the partition will have their system interface controls updated such that they ignore the outgoing box.

SOFT CONFIGURATION

Soft configuration is a software function that allows control over how the hardware resources are partitioned into systems and how each system's memory is partitioned into address spaces. Since the B7900 hardware has much greater architectural flexibility than any previous machine, extensive changes to the soft configuration mechanism were required.

The SYMBOL/CONFIGURATION file is the human interface to the soft configuration mechanism. This file is created or modified to define the possible configurations of the hardware resources. The SYMBOL/CONFIGURATION file is converted into a SYSTEM/CONFIGURATION file by the utility program SYSTEM/CONFIGURATOR.

SYSTEM/CONFIGURATION is the MCP's interface to the soft configuration mechanism. This interface is established through the ODT CF command. Once these files have been created, and the SYSTEM/CONFIGURATION file has been attached through the CF command, actual reconfiguration is done through the ODT RECONFIGURE command. Reconfiguration can also be performed from the Loader by using the LOAD CONFIGURATION and RECONFIGURE commands. The syntax and semantics of the CF and RECONFIGURE commands are specified in the Operator Display Terminal (ODT) Reference Manual (Form # 5011687).

DATA COMMUNICATIONS

The B7900 Data Communications Subsystem is substantially different at the hardware level from B7700 and B7800 (DCP) data communications. In place of the DCP and Adaptor Cluster modules, the B7900 uses the Network Support Processor (NSP) and Line Support Processor (LSP) modules.

The NSP and the LSP are Data Link Processors (DLPs) that communicate with the system through the HDU. An LSP can support up to 16 full or half-duplex data comm lines, and an NSP can control up to eight LSPs with a maximum of 96 lines.

A Network Definition Language, called NDLII, was developed for use with this new hardware. NDLII is described in the B5000/B6000 Series NDLII REFERENCE MANUAL (form no. 5011828). This document also describes the logical partitioning of data comm functions between the NSP and the LSP.

The SOURCENDLII symbolic provides an example of how NDLII programs are written. In order to provide a compatible interface to an MCS, certain NDLII programming conventions must be followed. The SOURCENDLII symbolic contains numerous comments explaining these conventions in detail.

In order to initialize an NSP, a NIF file and firmware file must be provided. A standard firmware file called FIRMWARE/NSP is provided. The NIF file is provided by the user by compiling a source NDLII program, such as the example SYMBOL/SOURCENDLII. The file name of the NIF must have "NIF" as the last node of its identifier. If <prefix>/NIF is the NIF file, the MCP creates an auxiliary file <prefix>/DCPCODE when the NSP is initialized with the NIF for the first time.

The ID (Initiate Datacom) ODT command is used to initialize an NSP, interrogate or alter the setting of various data comm options and direct commands to the DCCONTROL process.

One of the functions that can be initiated through the ID command is data comm auditing. A program, SYSTEM/DCAUDITOR, can be used to analyze the files produced during the auditing process.

A dump of the local memory of a specified NSP can be requested through the DUMP option of the ID command. The file produced can be analyzed with a program, SYSTEM/NSPDUMPANALYZER.

A report of the network configuration can be obtained through the DCSTATUS utility.

B5900 HARDWARE OVERVIEW

The B5900 central processing unit represents an architecture based on a series of fully microprogrammed function processors supported by a multi-bus structure. Despite the simplicity of its internal operation the B5900 is fully code compatible with all other large systems.

The processor is divided into five main areas that operate concurrently. These modules are serviced by two high-speed communication buses that are are capable of simultaneous data transfer (see figure 1-21).

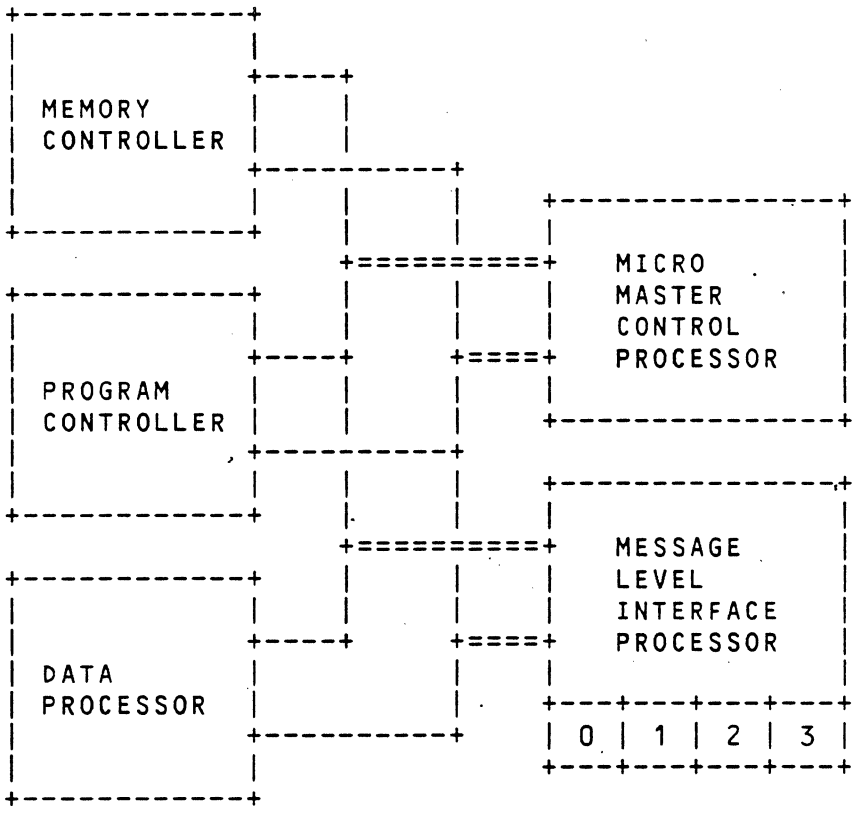


Figure 1-21. B5900 System

BUS STRUCTURE

There are two major buses on the B5900, the Main Data Bus (M-Bus) and the Control Bus (C-Bus). The M-bus is 52-bits wide to allow the movement of one word (48 data-bits, 3 tag-bits and 1 parity-bit) through the system at high speed. The C-Bus is 30-bits wide and is used by the Micro Master Control processor to control the other modules in the CPU.

MICRO MASTER CONTROL PROCESSOR

This is the master processor designed to execute the large system instruction set. It is a micro-programmed processor that guides the other processors through the execution of an instruction. This is done by addressing commands to a specific CPU component via the C-Bus.

PROGRAM CONTROLLER

This processor is designed to independantly decode instructions for the Micro Master Control processor as well as work directly with the Memory Controller to fetch additional program words.

DATA PROCESSOR

This unit is responsible for the actual arithmetic and logical processing done on the B5900. It is fed a series of commands from the Micro Master Control processor via the C-Bus.

MESSAGE LEVEL INTERFACE PROCESSOR

This unit provides an interface between the CPU and the DLPs actually doing the I/O. It will manage up to four MLI ports and is responsible for providing the DLPs with a high speed path to and from memory.

MEMORY CONTROLLER

The Memory Controller is responsible for providing the system with access to up to 6.2 MB of memory. It also provides the pathway to GLOBAL(TM) Memory.

B6900 HARDWARE OVERVIEW

The Burroughs B6900 system is high speed computer system derived from the B6800 series architecture. It was the first large system to support Universal I/O (UIO) hardware. It provides reliable high speed processing with a relatively low environmental cost in either floorspace or power.

The system is divided into three major parts, each operating to provide independant services to maximize system throughput. A schematic of the B6900 is shown in figure 1-22.

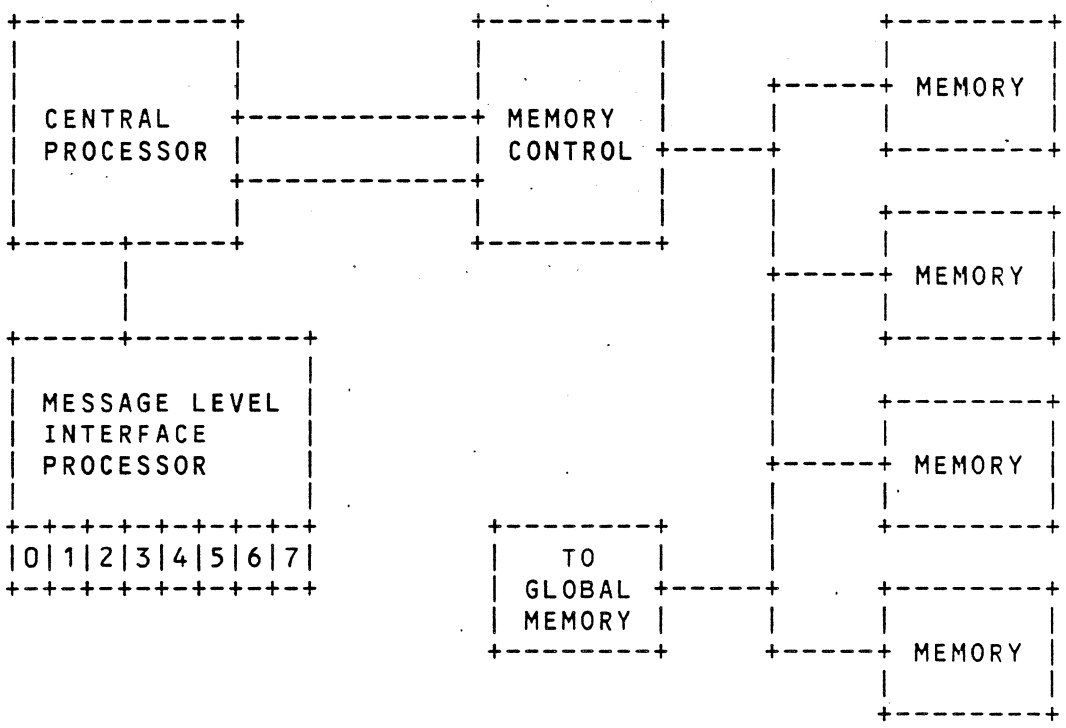


Figure 1-22. B6900 System

CENTRAL PROCESSOR

The Central Processor is the unit responsible for all program and MCP code execution. It does all arithmetic and logical operators based on the operator strings it is directed to execute. The CPU itself is almost identical with the B6800 CPU. A two words top-of-stack buffer is maintained in the CPU to speed operation. Also there is an automatic 1-word code look-ahead that is done via the CPU's second port to the Memory Control. This second path is dedicated to the look-ahead function.

MEMORY CONTROL

This unit provides the CPU with access to up to 6.2 MB of high speed memory. It acts as an exchange to allow either of the two CPU memory request ports with access to up to five memory modules. It is the fifth memory module that is the B6900's path to GLOBAL(TM) Memory.

MESSAGE LEVEL INTERFACE PROCESSOR

This processor is responsible for all the management of I/O done to and from the DLPs. It handles all main-memory queue manipulation as well as provides a path to memory for the DLP when data transfer is required. The MLIP must share the CPU's data path to memory, however, it is designed to work in a "burst" mode so as to make the memory accessing as efficient as possible. The MLIP is capable of handling up to 8 MLI ports. All input/output, both peripheral and datacom, is managed by the MLIP independantly of the CPU.

B5900/B6900 I/O OVERVIEW

This section deals with the technical aspects of the B5900/B6900 Universal Input Output (UIO) hardware operation. The details of both the hardware and software operation of UIO are extremely well covered in the manual "OVERVIEW OF PHYSICAL I/O ON B5900/B6900 SYSTEMS" (form #5013758). This section will simply review the main concepts behind UIO.

MAIN COMPONENTS

Below are listed the main components of the UIO system hardware:

1. Message Level Interface Processor - That part of a B5900/B6900 mainframe that is given the task of managing the I/O by the central processor. Its duties will include I/O queue handling, path queue handling, data transfer to and from memory, DLP initiation and generating an interrupt to flag the CPU to handle all queued results.

2. Message Level Interface - A standard hardware/software protocol to allow the development of an I/O subsystem that is independant of the characteristics of a given mainframe.

3. Line Expansion Module - A device to allow a Message Level Interface port to connect to up to 7 I/O Bases rather than just one.

4. I/O Base - A physical card cage to house up to 8 DLPs.

5. Data Link Processor - A specialized microprocessor with sufficient intelligence and storage to complete 1 entire I/O to a given peripheral. DLPs are designed to interface to specific peripheral devices and represent the connection port of the mainframe to the peripheral.

I/O INITIATION

The general flow for I/O on a UIO system is as follows:

1. The CPU formats an IOCB (I/O Control Block). This data structure totaly defines the nature of the I/O to be done including unit number and location, I/O control bits, buffer location, I/O queue locations and reserved space for I/O results and times.

2. The processor executes the CUIO operator passing it a descriptor to the IOCB.

3. The MLIP links the IOCB into the unit's command queue.

4. If this is the first I/O in the command queue then the unit's DLP is checked to see if the I/O can be initiated. If it can, the IOCB is delinked from the command queue and the DLP is passed the IO to initiate.

5. If the unit is sharing a path with other units and the DLP is busy then the MLIP will link the unit's entire command queue into the specified horizontal queue. The command queue will remain in this state until some other unit on the same path completes its command queue, freeing the DLP.

6. When the MLIP senses that an I/O has completed, it links the IOCB into the designated result queue. If the interrupt bit is on in the IOCB's MLIP control word, or if an I/O exception occurred or if the command queue is empty then the CPU will be given an I/O finish interrupt. If the command queue is empty then the MLIP will also check the unit's horizontal queue to see if there is another command queue queued for activation.

7. Upon receipt of an I/O finish interrupt the CPU will process the entire result queue. For each IOCB the following will be done:

   - delink from result queue
   - perform exception handling if necessary
   - perform all statistical accounting
   - wake up any task waiting on the I/O's completion

SECTION 2.

NEWP

INTRODUCTION

NEWP is documented in a file named NEWP/SUMMARY on the DOCUMENTS tape. This section is composed of comments and additional information on the NEWP language. The structure of this section is parallel to that of the NEWP document. Within this section all section names refer to the NEWP document.

The discussion of NEWP is followed by some programs which demonstrate problems a NEWP programmer could have. This is followed by notes on MCP compilation.

PROGRAM STRUCTURE

Blocks which are not procedures are not entered with an ENTR operator in NEWP (there is no MSCW or RCW). Items which are declared in the block are simply added to the stack at the current lex level.

If an array is declared in a block (which is not a procedure) an SCW and call on BLOCKEXIT will be generated. The SCW will instruct BLOCKEXIT to clear the block of any arrays.

Procedures which have arrays declared will generate calls on BLOCKEXIT. The call on BLOCKEXIT is the last thing in the procedure. To avoid BLOCKEXIT the programmer may release the memory area for any arrays and do an EXIT. If the procedure is typed the programmer should use the RETURN statement.

PARAMETER PASSING

The following is a list of 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

STATEMENTS

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.

DATA TYPE	OPERATION 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

REPLACE STATEMENT

The following is a feature of the large systems hardware. 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;
```

DESCRIPTOR DECLARATION

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.

INTRINSICS

UNLOCK [MISC]

The code emitted for the UNLOCK statement will do a B7800 STOREQ purge; i.e., UNLOCK(R) produces the code: NAMC<R>, ZERO, STON, INCN. This code will be emitted if the B7000 compiler control is set.

SETLIMITS [MACHINEOPS]

With MOD3 MCM bits [7:8], bits [13:6] and bits [19:6] of the first 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 first parameter represent the availability mask, the upper memory addressing limits and the lower memory addressing limits respectively.

SUSPEND [MACHINEOPS]

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.

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 on events
- 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];
    ARRAY Z[0:4];
    Z[0]:=A[0];
 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.
```

IRW LOOP:

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

MCP COMPILATION

The MCP consists of several different programs that must be compiled separately and then bound together. The compilation of system software is discussed in the System Software Operational Guide (SOG) Volume 2 section 14 (form 5011679). The SOG contains a detailed description of the procedure and sample JOB decks.

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. SEPCOMP is documented in the NEWP document. 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
   $# MCP THE PATCHES
   $ 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 COMPILE";

   REMOVE PATCH/NEWPSEP;
END JOB.
```

SECTION 3

SYSTEM INITIALIZATION

INTRODUCTION

This section discusses the different ways a Large System may be initialized. This includes a discussion of entry into the SYSTEM/LOADER and some MCP procedures. Two MCP procedures, GETITGOING and PRIMARYINITIALIZE, are responsible for nearly all system initialization.

INTRODUCTION TO SYSTEM INITIALIZATION

Large Systems can be initialized in several ways. The form of initialization selected depends on the degree of initialization required. The types of initialization are:

COLD START

COOL START

WARM START

CHANGE MCP (CM)

HALT/LOAD

Each of these topics will be discussed with the purpose of outlining the differences and effects of each type of initialization.

COLD START

When a COLD start is performed four major functions are done. A COLD start builds a disk label and directory on the HALT/LOAD unit, loads a copy of the MCP to disk, builds MCP tables (and a BOOTSTRAP) and performs a HALT/LOAD to the loaded MCP. Some of the MCP tables loaded are outlined at the end of this sub-section.

A COLD start is performed by running the LOADER. The input is COLD start information which may be entered as a card deck, ODT transmissions or a data file on disk or tape. More information on the LOADER can be found in the System Software Site Management Manual (form number 5014418).

A COLD start must be performed anytime the HALT/LOAD structures (MCP tables) or LABEL BLOCK are damaged on the HALT/LOAD unit. The LOADER will never do a COLD start without operator approval.

COOL START

A COOL start loads a new MCP to disk and updates existing MCP tables. The LOADER then performs a HALT/LOAD to the new MCP. A COOL start is performed by running the LOADER. The input is a modified COLD start deck.

The major difference between a COLD start and a COOL start is that during COOL start the disk directory is not destroyed or rebuilt. A COOL start could be used to replace a MCP that was bad.

WARM START

A WARM start updates MCP tables and performs a HALT/LOAD to the MCP. In a WARM start only the MCP tables are updated. A WARM start is performed by running the LOADER. The input to the LOADER is a COOL start deck without the LOAD card.

The basic difference between a COOL start and a WARM start is that during a WARM start a new MCP will not be loaded. A WARM start could be done to alter the UNIT cards on the B7800 or load a new PCD on a B7900 (this could be required if someone had reconfigured to a PCD that would not HALT/LOAD). A WARM start can also be used to load disk pack firmware.

CHANGE MCP

A Change MCP (CM) can only be used on a running system and is used to change the current MCP. The CM command can also be used to create new HALT/LOAD families, delete (unmark) HALT/LOAD families, interrogate the next MCP to be used and cancel a pending CM action. A CM # can be used to transfer control to a new MCP without updating the bootstrap. A CM is initiated by entering the CM command at the ODT. The syntax for the CM command can be found in the ODT reference manual.

When a CM is done to the HALT/LOAD family no action is taken

until a null mix occurs; at that time a HALT/LOAD to the new MCP will be executed. If a CM was done to create a new HALT/LOAD family, the bootstrap and MCP structures will be written on the family. A manual HALT/LOAD on the family will be required to use the new HALT/LOAD family.

HALT/LOAD

A halt/load is the action taken to reinitialize all main-memory code and data structures used by the MCP. Only information contained in the MCP Structure Tables (parameter and configuration information) and the JOBDESC file (job management and restart information) is retained across a halt/load. All other main-memory information is rebuilt to allow a clean start in terms of system operation.

The halt/load requires that a disk pack has been prepared with the necessary information. This halt/load conditioning can be done offline by SYSTEM/LOADER or done online by the CM ODT command. A valid halt/load disk pack will contain, at a minimum, the following data structures:

- valid disk pack labels
- valid disk pack directory
- MCP code file (contained totally on the H/L disk pack)
- bootstrap code or bootstrap pointers to MCP code file
- MCP parameter and configuration tables

The specifics of the halt/load process will differ with the hardware of each Large System. The following general outline is characteristic of all machines. The detailed flows are given at the end of this section.

1. DISK-LOAD or maintenance processor

   The hard-electronics or the maintenance processor will load and execute a small bootstrap program starting at word 0 in memory.

2. BOOTSTRAP program

   The BOOTSTRAP program uses information encoded in the bootstrap area (first 3 segments of the halt/load diskpack) to read the code for the MCP procedure GETITGOING from disk into memory. The PCW for GETITGOING is placed at D0+3 and its segment descriptor is placed at D0+1. GETITGOING is then entered.

3. GETITGOING

   This routine is designed to complete the bootstrap process of loading the MCP code file. It reads in the following information from the halt/load disk pack.

- disk pack label
- disk pack directory
- MCP Structure Tables (MCPINFO, etc)
- MCP code file diskfile header
- segment 0 of MCP code file
- DO stack image
- MCP initialization and save code

The order of reading is dictated by GETITGOING bootstrapping from one piece of information to the next. GETITGOING calls MCP procedure PRIMARYINITIALIZE to complete initialization.

4. PRIMARYINITIALIZE

The H/L processor brings all processors and I/O processors in the configuration online. Eight halt/load stacks are placed in the stackvector by the H/L processor. After some initialization all processors are directed to execute code to move to their own halt/load stack and invoke the MCP routine SECONDARYINITIALIZE.

5. SECONDARYINITIALIZE

This routine does the major initialization of MCP data structures. All memory is verified and all required MCP arrays and tables are initialized. SOPHIA, BOJEOJ and FIBSTACK are called to do their one-time-only initialization. AREAMANAGER, CONTROLLER and STARTSYSTEM are FORKed. The slave processors are released to move to their IDLER or ETERNALIR stacks and begin normal multiprogramming. The H/L processor moves to its ETERNALIR stack and completes initialization by calling the MCP routine MERGER.

6. MERGER CALL IN ETERNALIR

This routine is called once by ETERNALIR when it first BOJs. MERGER returns all memory areas associated with initialization. ANABOLISM is FORKed and then called so that all MCP routines thus far FORKed will be initiated into the mix (including anabolism itself).

MCP TABLES

Some MCP information is saved in the structure tables. There are two structure tables (primary and secondary). These tables point to the actual location of the data. This data includes MCPINFO, UNITADDL, PCTABLES (PERCONINFO), SLFUNCTIONTABLE, CONFIGURATION information and SWAPPER information.

The UNITADDL table provides information about peripheral

status. This status includes which units are saved or protected. The table is built at COLD start time and altered by the MCP.

The MCPINFO table provides information which is to be saved across HALT/LOADS. This information includes run time option settings, OLAYROWSIZE, last MIX number, system serial number, date, DCP prefix and other information. The table is built at COLD start time and altered by the MCP.

The PCTABLES provides information to initialize unit tables for the IOM. The information comes from the UNIT cards provided to the LOADER. This table is rebuilt during each run of the LOADER.

B6800 SYSTEM INITIALIZATION

On the B6800 SYSTEM/LOADER is run via a process bootstrapped through the card reader. A card deck, produced from the compilation of the NEWP program UTILOADER, is placed in the card reader.

The CARD-LOAD select switch is set to an ON position and a halt/load is performed. The CPU is idled and the multiplexor reads one card from the reader into word 0 of memory. At the completion of the card I/O the CPU is interrupted into the code just loaded.

The first card is a B6800 program designed to read in the next two cards. This 3-card loader then reads in the NEWP standalone program UTILOADER from the card reader into memory starting at word 0. When the load is complete the CPU is interrupted into the outer block code of UTILOADER.

The UTILOADER reads one card from the card reader in the format:

<tapename> <filename>

UTILOADER will then scan all available tape drives looking for a library maintenance tape named <filename>/FILE000. When it is located UTILOADER will load the NEWP standalone program contained in file <filename> from tape to word 0 in memory and will interrupt the CPU into the outer block of this new program. This program will be SYSTEM/LOADER.

B5900, B6900 SYSTEM INITIALIZATION

On the B5900 and B6900 series of large system, SYSTEM/LOADER is bootstrapped via the maintenance processor.

When powered up the maintenance processor executes a ROM based routine to load the maintenance processor operating system from floppy disk. On the B5900 a menu is presented and the operator selects the BOOTLOAD option to load the NEWP standalone program UTILOADER from floppy disk to word 0 in main memory. On the B6900 the maintenance processor will accept a command from the operator to load the NEWP standalone program from the SYSBOOT tape to word 0 of memory. The CPU is then interrupted into the outer block code of UTILOADER.

The UTILOADER then presents a menu of functions. The operator selects TAPELOAD and UTILOADER will then request <tapename>, <filename>, unitnumber and path information. When entered and validated UTILOADER will load the NEWP standalone program contained in the file <filename> from the library maintenance tape <tapename> to word 0 of memory. The CPU is then interrupted into the outer block of the new program. The program loaded will be SYSTEM/LOADER.

B7800 SYSTEM INITIALIZATION

System initialization consists of bringing the MCP 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 3-1. The path on the right of the figure assumes a valid MCP is on disk (a halt/load is performed). If a valid MCP is not on disk or there is no valid directory on disk, then the left path must be traversed, followed by entry into the disk boot.

System initialization begins when the operator presses the enable and load buttons on the operator control console (figure 3-2). The hardware takes over the initialization process when the operator releases the enable button. Depending on the state of the operator's console, either the card load or the disk path will be taken.

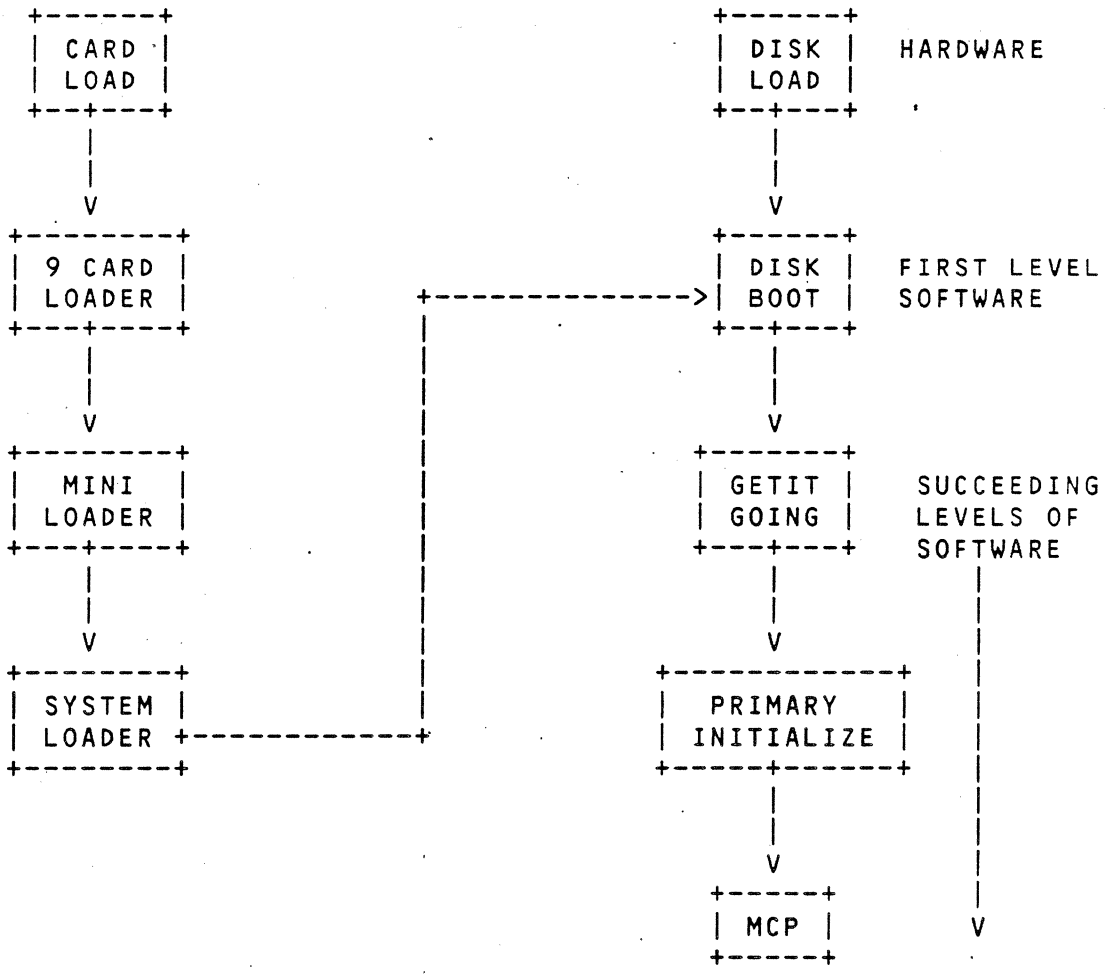


Figure 3-1. Bootstrap Organization

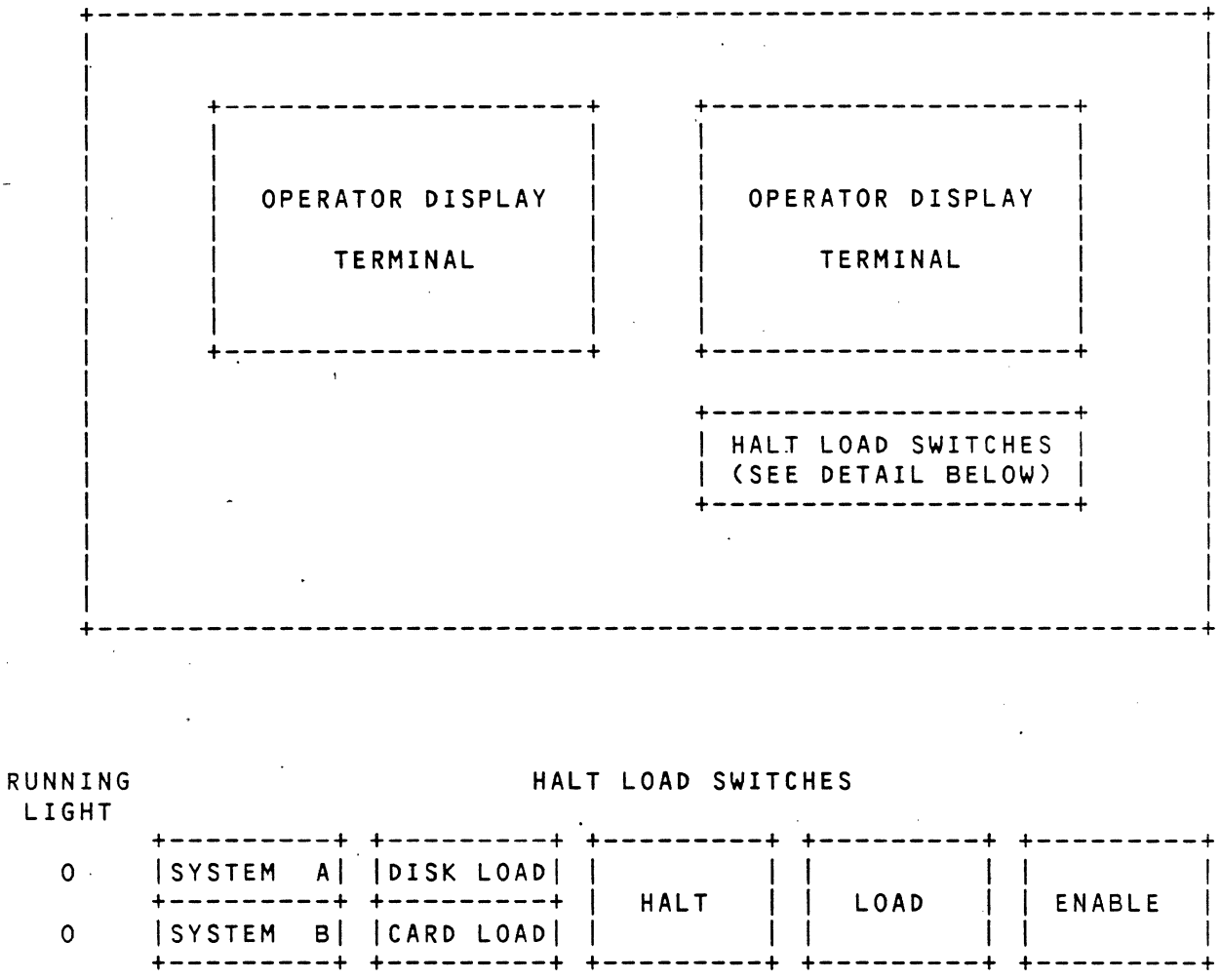


Figure 3-2. Operator's Control Console

The COLD START/HALT LOAD SELECTION CARD (figure 3-3) 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. 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 module number of the processor that will take part in the initialization process. The other switches are used to select a card reader channel (card load operation) or disk channel and unit number (disk load operation).

Figure 3-4 shows the format of the three most important words used by the system during the hardware and first level software portion of system initialization.

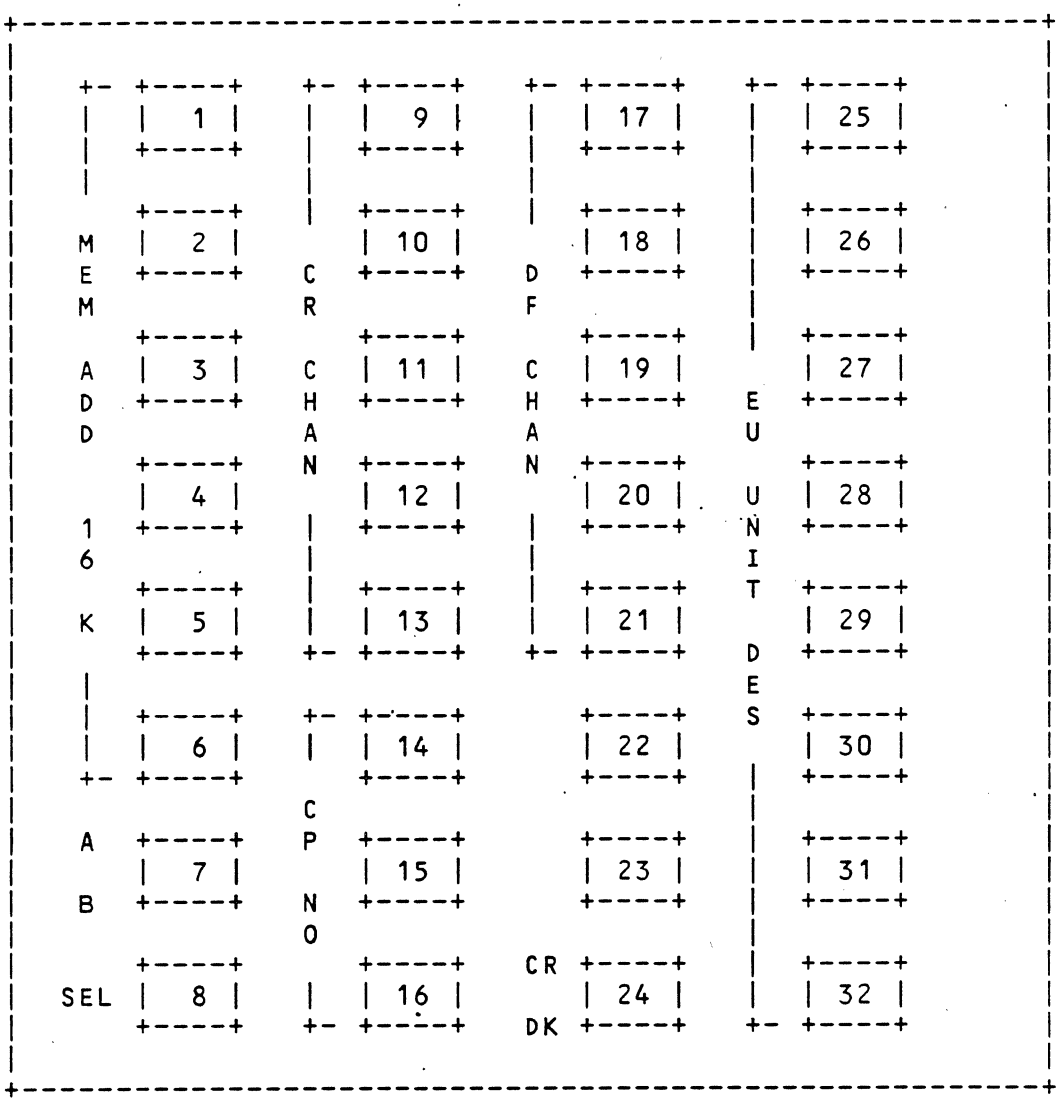
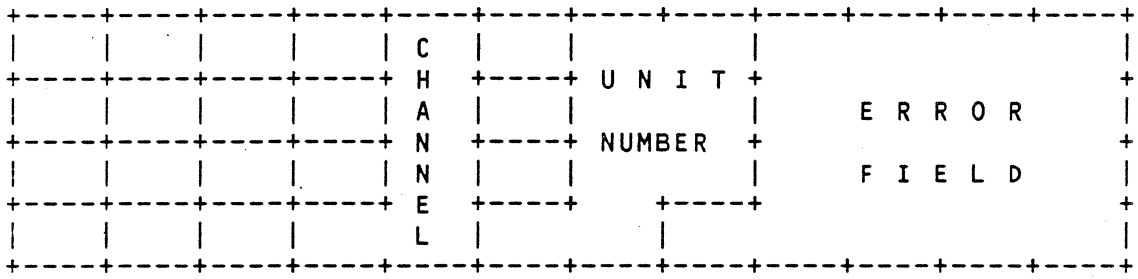
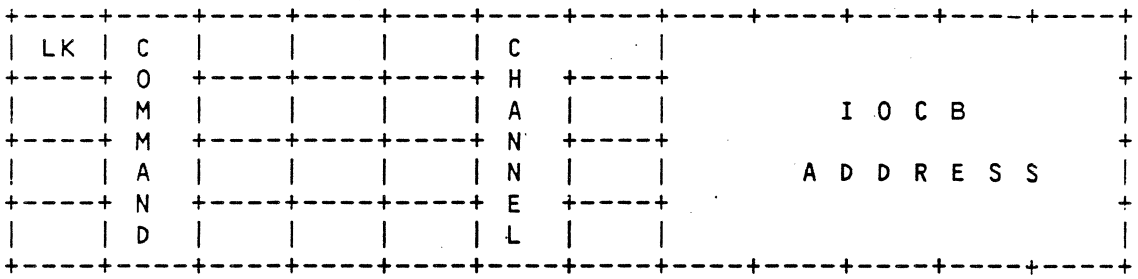


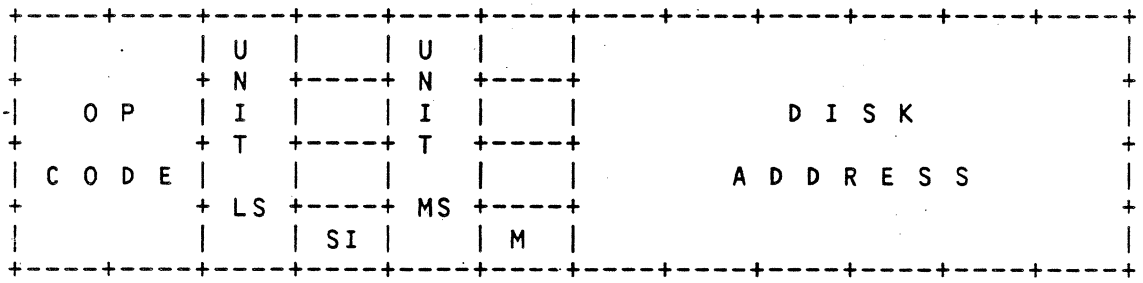
Figure 3-3. Cold Start/Halt Load Selection Card



HARDLOAD RESULT DESCRIPTOR AT MEMORY[4]



SYNC I/O COMMAND WORD AT HOMEADDRESS[0]



DISK CDL AT IOCB[4]

Figure 3-4. Disk Boot Words

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 (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. In both cases the CPM is interrupted into the loaded code. If there are any unexpected errors in either of these hardware functions the operator must start the operation from the beginning.

The 9-CARD loader (read by the hardware) then reads the NEWP standalone program MINILOADER from the card reader into word 0 of memory. The CPM is then interrupted into the outer block code of MINILOADER.

The MINILOADER will read one card from the card reader containing the following information:

<CH>/<UNIT> <filename>

MINILOADER will then read a library maintenance tape on tape channel <CH> and physical unit number <UNIT>. MINILOADER will load the NEWP standalone program <filename> from tape to word 0 of memory and then interrupt the CPM into the outer block of this program. The NEWP program that is loaded is SYSTEM/LOADER.

The SYSTEM/LOADER is responsible for constructing the directory, loading the specified MCP from tape to disk, and entering the MCP via the disk boot. An outline of the MINILOADER and SYSTEM/LOADER is detailed on subsequent pages.

Once the disk boot is entered (from the hardware or SYSTEM/LOADER), it reads in GETITGOING. This procedure and the procedures it calls are outlined at the end of this sub-section.

B7900 SYSTEM INITIALIZATION

The program which executes on an MTS2/MIP pair to provide for configuration and system initialization of B7900 partitions is called SYCON (SYstem CONsole). SYCON also provides "panels" for each of the B7900 box types (CPM, MSM, HDU, AP); these panels are oriented towards operating the system, rather than towards detailed diagnostics of hardware failures. The SYCON software (and related files) are loaded from the ICMD drives packaged in the "leg" of the B7900 console. SYCON is a "menu-driven" program; it displays various screens, on which the user may select pre-defined functions.

SYCON has access to all boxes in a system. However, it is constrained both by hardware and by software convention to limit access to fit within the overall system mainframe partitioning scheme. Each box contains a Partition Identification Register (PID). PID is not used by the hardware, but is used by software to designate the way the box is being used; ie, the "state" it is in. INHIBIT and AVAILABLE are two such states. SYCON and the MCP are the primary softwares that manipulate the PIDs, and this is done during any partition configuration changes.

MINIMAL CONFIGURATION

The Minimal Configuration is a list of mainframes and peripheral devices which is provided to primitive system software which does not have access to or knowledge of the Peripheral Configuration Diagram. This software includes the Utiloader, the Loader, the Boot and certain primitive portions of the MCP (GETITGOING and Standalone Tape Dump). Included in the Minimal Configuration are the H/L HDU id, the H/L CPM id, the H/L MSM id and the Device Numbers and paths for the H/L ODT, H/L tape unit, H/L disk pack unit and H/L printer.

The Minimal Configuration is specified by the operator at the Maintenance Console (using SYCON) and is maintained in the H/L HDU's RAM. The Mainframe Ids must be specified when the partition is initially established at the Maintenance Console.

It should be noted that the system software has no way of verifying the device numbers for the peripherals in the Minimal Configuration until MCP peripheral initialization is complete. The path information will be used to perform the I/O operations and the device numbers will be used for display purposes only.

It is strongly recommended that the Minimal Peripheral Configuration be fully specified at the Maintenance Console, but it is not a requirement. There are several operator interfaces for displaying and modifying this information. The Utiloader has a CONFIGURATION command and allows the tape unit and diskpack unit to be specified or changed. The Loader will display the configuration and will allow the disk unit and the printer to be specified or changed. In addition, the Loader will allow the paths to another tape or disk unit to be specified for use during a specific Loader command. The MCP will display and modify the configuration information via the BOOTUNIT command.

In order to avoid confusion, all software uses the same displays and terminology when referring to the units in the Minimal Configuration.

BOOT

The BOOT is a small NEWP standalone program which, when executed, will initiate a Halt/Load by entering the MCP. There are 256 words reserved in the HDU's RAM which will contain the BOOT and the Minimal Configuration information required to initialize the system. Words 0 to 245 are reserved for the BOOT's code and D0 stack environment. Words 246 to 255 are reserved for the Minimal Configuration information which is used by the BOOT, the UTILOADER and the LOADER.

Word	Contents
246	Unit number for the Halt/Load disk unit
247	Address word for the Halt/Load disk unit
248	Unit number for the Halt/Load ODT
249	Address word for the Halt/Load ODT
250	Unit number for the Halt/Load tape unit
251	Address word for the Halt/Load tape unit
252	Unit number for the Halt/Load printer
253	Address word for the Halt/Load printer
254	Halt/Load MSM id (used by hardware)
255	Halt/Load CPM id (used by hardware)

The address words contain the DLP address, the unit position, the HDU id and a validity bit. Words 254 and 255 also contain a validity bit. The BOOT is loaded into the HDU's RAM by SYCON when the HDU is powered up.

Entry into the BOOT is initiated via the LOAD command from SYCON or programmatic Halt/Load from the LOADER or the MCP. The MSMs and requestors are halted (if necessary) and put in an initial state. The CPM initial state has D0=0, F=0, S=4'2000', DENR=0 CENR=0. The processor is idle and will enter the BOOT using the PCW at D0+3 when interrupted by the HDU. The HDU initial state has the Home Address Queue Pointer = 16 (4'10'), the QM Environment Number = 0 and the Error Command Queue Pointer = 0. SYCON will initialize the mainframes by sending a maintenance Execute To Initial State (ETIS) to each box, setting up the Box Id registers and setting up the Halt/Load MSM to validate the Halt/Load page (page 0 of environment 0). The software will initialize the mainframes for a programmatic Halt/Load by sending the ETIS hardware interrupt to each processor and HDU. Word 256 in memory is then initialized with Halt/Load information, and the Halt/Load HDU is issued the LOAD hardware interrupt, which causes it to send the ETIS hardware interrupt to the Halt/Load CPM, store the contents of the HDU RAM at address 0 of the Halt/Load page and send a software interrupt to the Halt/Load CPM.

The BOOT obtains configuration information from the HDU's RAM

which has been stored in memory. The path to the Halt/Load disk unit and the Halt/Load HDU id are in word 247, and the Halt/Load MSM id is in word 254.

The BOOT obtains MCP information from the Halt/Load disk. Sector 3 of the Halt/Load disk contains MCP code file information for three MCP entry points. Currently only two of them are used. One is used for GETITGOING information for a normal Halt/Load and for an SATAPEDUMP. The other entry point is used for GETITGOING information for a memory only CM (CM#). The MCP code file information for each entry point consists of GETITGOING's binary disk address, length, and PIR and PSR. Word 256 in memory indicates which entry point to use and contains informaton to be passed on to GETITGOING which includes a bit to invoke SATAPEDUMP and a bit to indicate if the Halt/Load reason is valid in memory.

The BOOT will set up its I/O structures (HAQ, CQ, RQ, IOCB), and perform eight I/O operations. It will enable the HDP and MLI port in the path to the disk unit, clear the base, enable and clear the disk DLP, read in the MCP code file information and GETITGOING from disk, and enter GETITGOING. GETITGOING will reuse the BOOT's I/O structures.

The BOOT does not set up an Error CQ or update the Error CQ Pointer in the HDU. This will cause the HDU to update its Fault Buffer and return to idle if an error should occur for which an Error IOCB will be generated.

OUTLINE OF B6800 BOOTSTRAP CODE

The B6800 bootstrap program resides in the first 30 words of the halt/load diskpack. It was placed there either by SYSTEM/LOADER or the MCP routine CHANGEMCP (CM ODT command). This bootstrap program originates from hand-coded machine operators in a value array in each utility. Prior to being written to the diskpack it is customized with GETITGOING's disk-address and length.

When the HALT and LOAD buttons are pressed with the CARDLOAD select switch OFF the MPX reads from segment 0 on the halt/load diskpack (specified by an ID jumper block in the CPU) to word 0 of memory. The CPU is initialized to an IDLE state with the following register settings:

```
D0  = 0
PDR = 4
S   = 2000
LL  = 0
PBR = 1
```

The I/O complete interrupts the CPU into the bootstrap code.

1. Hardwareinterrupt code at 0006:0 gets the result descriptor for the H/L read and returns.

2. Hardwareinterrupt code returns the result descriptor in top stack to code at 0001:0. This code disables external interrupts and branches to 0008:1.

3. Save the result descriptor from H/L read at (0,16).

4. Set S = 4'3E00' + PROCID*4'4000'.

5. Bring CMINFO from (0,2) to top of stack.

6. Get H/L unit number from H/L result descriptor. Loop infinitely if zero.

7. Build SCANOUT initiate-IO literal with unit number.

8. SCANOUT to initiate a read using the pre-formatted IOAD at (0,5) and the pre-formatted IOCW at MEMORY[15]. Note: The disk address and length of GETITGOING has already been set up in these control words.

9. IDLE until interrupt. Hardwareinterrupt code returns the result descriptor when the I/O is complete. Loop infinitely if an I/O error is detected.

10. Move GETITGOING down from MEMORY[16] to MEMORY[0]

overwriting the bootstrap program. Note: The last part of the bootstrap program will execute properly because it resides in the P register and not in memory.

11. Restore CMINFO at (0,2) from the copy saved in the top of stack.

12. Force an INVALID-OP fault into GETITGOING.

OUTLINE OF B7700/B7800 BOOTSTRAP CODE

The B7700/B7800 bootstrap program resides on the first 30 words of the halt/load diskpack. It was placed there either by SYSTEM/LOADER or by CHANGEMCP (CM ODT command). It originated from a value array of hand-coded machine operators. The bootstrap program was customized with GETITGOING's disk-address and length prior to being written out.

When the HALT and LOAD buttons on the B7700/B7800 console are depressed with the LOAD-SELECT switch in the DISK LOAD position the H/L IOM (designated by IOM selection card) reads from segment 0 of the H/L diskpack (designated by the H/L IOM selection card) to memory word 0. The I/O that is done is a special type of SYNC-IO and the I/O result is placed at word 4 of the IOMs home address area (HA initialized to 0). The H/L CPM is set to a PAUS state and initialized with the following register values:

```
S   = 4'2000'
PIR = 8
PBR = 0
PSR = 0
```

The H/L CPM (designated by the H/L IOM selection card) is interrupted by the I/O complete and begins normal execution.

1. Set processor registers:

```
D0   = 0
PBR  = 0
S    = 4'1E00'
BOSR = 0
SNR  = 0
```

2. Mark the stack and do a NAMC on (0,3).

3. Get HA[0] (at (0,0)) and update it with the H/L channel number from the HARDLOAD result in HA[4] (at (0,4)). Store it back in (0,0).

4. Get CDL from (0,7) and update it with the H/L unit number from the HARDLOAD result descriptor. READLOCK the CDL back to (0,7) to store it and purge ASM.

5. Read up interrupt request register and use it as the mask to interrupt the H/L IOM via the interrupt channel operator. This will cause the IOM to do the SYNC-IO read of GETITGOING as defined by the HA command just formatted. Note: The disk address and length of GETITGOING was pre-formatted into the CDL and BUFFER-DESCRIPTOR.

6. PAUS until interrupt. Look at SYNC-IO result descriptor at HA[5] (at (0,5)). Loop infinitely if there is an I/O error.

7. Generate a valid DO MSCW and store it at (0,0).

8. Do an enter. This invokes the PCW at (0,3) that now points to GETITGOING code in segment 1.

OUTLINE OF B5900/B6900 BOOTSTRAP CODE

The B5900/B6900 bootstrap program resides at the base of memory (word 0) for a running system. It is placed there by the UTILOADER via the HALTLOAD or LAYDOWNBOOT command. The bootstrap program is simply the code segment BOOTIOREQUEST from the UTILOADER code file. When the code is moved to low memory it is customized to identify the H/L diskpack in MEMORY[6]. This field can be updated in a running system via the BOOTUNIT ODT command.

When a B5900 is CTRL-Hed the maintenance processor presents a menu to the operator. If the LOAD option is selected, the maintenance processor clears the CPU, sets DO to zero and forces an interrupt to interrupt the CPU into the bootstrap program.

On a B6900 when the HALT and LOAD buttons are depressed, with the LOAD-MODE switch on, the maintenance processor clears the CPU, sets DO to a value of zero and causes an interrupt to interrupt the CPU into the bootstrap program.

When the UTILOADER command HALTLOAD is used on either a B5900 or B6900 the H/L diskpack number is requested. A new bootstrap is placed in low memory, DO is set to zero, S is set to 4'7F00' and an INVALID-ADDRESS interrupt is caused to interrupt the CPU into the bootstrap program.

1. Interrupt calls procedure BOOTHARDWAREINTERRUPT.

2. BOOTHARDWAREINTERRUPT calls procedure MAINPROGRAM if the interrupt was a INVALID-ADDRESS.

3. An ERROR-IOCB is initiated and an MLI master clear is done.

4. The HLUNIT number and path is retrieved from MEMORY[6].

5. If the halt/load is not a CM# then:

   a. Read the first 90 words of the H/L diskpack with tags.

   b. From the diskpack's bootstrap area get GETITGOING's disk-address, length and PIR/PSR.

6. Read GETITGOING from disk to MEMORY[4'6000'] forcing tag 3.

7. Cancel the error IOCB.

8. Set D0 to 4'7E00'.

9. Place CMINFO, MEMORY descriptor and GETITGOING segment descriptor in the new D0 stack.

10. Put GETITGOING's PCW at (0,3).

11. Fault into GETITGOING.

OUTLINE OF B7900 BOOTSTRAP CODE

The bootstrap program for the B7900 originates from the code file for the NEWP standalone program BOOT79. When SYCON initializes the HDU at power-up it loads this code from floppy disk to 256 words reserved in the HDU's RAM. All information regarding the Minimal Haltload Configuration is also kept in the last 10 words of this reserved RAM (H/L CPM, HDU, MSM, PK, SC, MT and LP). When the SYCON command LOAD is given, SYCON will cause the H/L HDU to dump this reserved RAM to MEMORY[0]. SYCON initializes the CPM as follows:

```
D0 = 0
S  = 4'2000'
F  = 0
```

SYCON interrupts the H/L CPM into the BOOT79 program.

1. Set up I/O structures (HAQ, CQ, RQ, IOCB).

2. Enable the HDP and MLI port in the path to the disk unit.

3. Clear the I/O base.

4. Enable and clear the disk DLP.

5. Read the first 90 words from the halt/load diskpack.

6. Get GETITGOING's disk-address, length and PIR/PSR from

the diskpack's bootstrap area.

7. Read GETITGOING from the H/L diskpack.

8. Do a procedure entry on GETITGOING.

OUTLINE OF MINILOADER

The 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. A detailed description of this process and figures of the stack can be found in the Software Operational Guide, Volume 2, Section 2 (form number 5011679). The following is a basic outline of the MINILOADER.

1. Read in the parameter card to determine where the tape unit is and what code file is required. A typical parameter card is: 1/80 SYSTEM/LOADER

2. Store the channel and unit number information from the parameter card and build a standard form name. The standard form name is described as follows.

   Character 1.

   Total number of characters in the string (self-inclusive).

   Character 2.

   Security byte:

   0. For MCP use.

   1. User file.

   2. System file.

   3. Usercode specified.

   Character 3.

   Number of identifiers in the file name.

   Character 4.

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

   For example the standard form name for SYSTEM/LOADER is:

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

3. Rewind the tape.

4. Space up the tape one tape mark which will place the tape read head immediately before the directory. The space operation is performed by building a CDL with a zero in the field which indicates the number of blocks to be spaced. This 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 which consists of one or more 901 word blocks.

   Each block consists of a one word transaction count and one word of link information. The link information is used 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 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.

   Additional documentation on library tapes can be found in the I/O Subsystem Manual Section B (form number 5001779).

6. Find the required file's location on the tape from the directory and 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. The save information is read into memory. The length and address of the information is obtained from segment zero of the code file. The information is read in TAG transfer format.

9. Rewind the tape.

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

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

12. Move the save information down to address 0.

13. 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.

OUTLINE OF SYSTEM/LOADER

The LOADER is normally brought into memory by the MINILOADER. Once the LOADER begins execution it starts reading cards to determine its objectives.

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 the B7800 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 process all cards until a "STOP" card is found. When the "STOP" card is read, the following is done:

    a. Build the flat directory and write MCP structures to disk.

    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.

OUTLINE OF GETITGOING

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

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

    a. Bootstrap.

    b. Open area.

    c. GETITGOING read in by bootstrap.

    d. Open area.

    e. Halt load processors stack.

2. Pick up information left by bootstrap (at MEMORY[2]). This information includes machine type, familyindex of H/L unit, memory only H/L (CM #) and path information.

3. Build MEMORY descriptor.

4. Change all PCW's in this stack to segment one. GETITGOING is not segment one but runs as segment one.

5. Set up hardware interrupt based on machine type. The interrupt routine can be used for memory testing. It will dead stop on most interrupts.

6. Check to see if a memory dump is requested. This is done by checking to see how long a conditional halt instruction takes. If it takes over 7 seconds the conditional halt switch is on (a dump is requested). If a dump is requested it will be taken.

7. Set up an initialization data area and code area above MOD one. This area is used to hold arrays and code used during initialization. The areas are found by trying to touch a memory MOD. If no interrupt occurs the MOD is present.

8. An initial STACKVECTOR is set up in the data area. An initial MCPSTACK (stack 8) is put in the STACKVECTOR. This MCPSTACK points to the base of memory.

9. An array is set up to save information that will be passed to other initialization procedures. Some information is placed in this array.

10. GETITGOING is moved to the initialization code area.

11. The H/L processors stack is moved to the initialization data area. The stack is placed in the STACKVECTOR (in location 0). GETITGOING and the STACKVECTOR are moved so the MCPSTACK can be read.

12. For UIO machines, initial I/O structures are set up. A B7800 can do a sync I/O, so it does not need these structures.

13. Read pack label from halt load unit. The pack label points to segment zero of the directory.

14. Read in segment zero of the FLAT directory. A pointer in segment zero is used to find the FLAT directory header. The FLAT directory header is read in (FLAT FLAT).

15. A pointer in segment zero is used to find the MCP structure table (primary and secondary). The structure tables are read into memory. These tables are a set of pointers (record numbers in FLAT) to structures such as MCPINFO, UNITADDL, CONFIGURATION and SWAPPER.

16. Read in MCPINFO.

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

18. Read in segment zero of the code file.

19. Read in DO stack image from MCP code file to memory at DOSETTING. Segment zero points to the DO stack image. The stack is read with a TAG transfer format. Memory links are built around the stack.

20. The proper stack number (stack 8) is placed in all PCW's in the MCP stack.

21. The DO stack is placed in the STACKVECTOR at MCPSTACK (stack 8). The base of the stack is initialized.

22. The proper segment descriptor for GETITGOING (the one in the MCP stack) is used. PCW's in the H/L processors stack must be changed (new SDI).

23. The STACKVECTOR is placed in the MCP stack (DO+2 is built).

24. Set the DO register to DOSETTING. DO points to the true MCP stack.

25. Place information generated so far in the DO stack.

26. Read in the structure Peripheral Configuration Diagram (PCD B7900) or Peripheral Configuration Table (PCTABLE B7800). Read in configuration information.

27. Read in the structures UNITADDL (unit information [saved, protected, mode read/write, reserved]), SWAPPERINFO and FUNCTIONTABLE (SL commands).

28. Check out memory log area which is used to log information during the initialization process.

29. Set up a LOCAL code memory area at low order memory for save procedures. A RESIDENT code area is already set up above the DO stack for save procedures.

30. Set up time stamp for MCP code file. Set up a few arrays (JOBDATA and backup queues).

31. 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. After the initialization procedure is invoked, the initialization procedures code is discarded. In addition, some of the procedures read in are based on machine type.

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

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

OUTLINE OF 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. This outline is of PRIMARYINITIALIZE70.

1. Make D0+3 point to the hardware interrupt routine in PRIMARYINITIALIZE. This routine is used for testing memory.

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

3. Generate MCM, IOM and CPM masks from configuration file.

4. All IOM's are told to load home address register (to 0). If the IOM will not load home address, it is removed from the IOM mask.

5. Code is placed at word 8 in memory. This code will set a few registers and enter a procedure called SLAVECPMSTARTUP. All slave (non-H/L) processors are interrupted. They will execute SLAVECPMSTARTUP.

   SLAVECPMSTARTUP sets a few registers and calls SLAVECPMHOLDER. SLAVECPMHOLDER sets a bit to indicate the processor is running (used to build processor mask). The processor then waits for the H/L processor to give it a command.

   At this point the processors are set up to talk to each other thru a communication cell. The H/L processor can place the command in the communication cell and interrupt a slave processor. The slave will read the communication cell and do what is requested. After the command is done the processor will wait for the next command.

   The types of commands are set time of day, set CMR register, move to another stack, map memory and check IOM/CPM interface.

6. The H/L CPM sets its CMR register and tells the slaves to set their CMR register. The CMR (Condition Mask Register) register is used to mask interrupts (if the bit is on the interrupt is not masked).

7. All processors set time of day register.

8. Procedure CONFIGESTABLISHER is called to establish MCM hardware. This procedure will see which MCM's are present, the model of the MCM and the limits of the MCM. Memory masks are generated for global and local boxes.

9. The Halt/Load stacks are set up (see figure 3-5). These stacks are set up with an environment which will allow the processors to enter SECONDARYINITIALIZE. The actual stack numbers used are based on the number of boxes on the system (B7800 4, B7900 46). Therefore, figure 3-5 shows how the stacks are linked but does not indicate proper stack numbers.

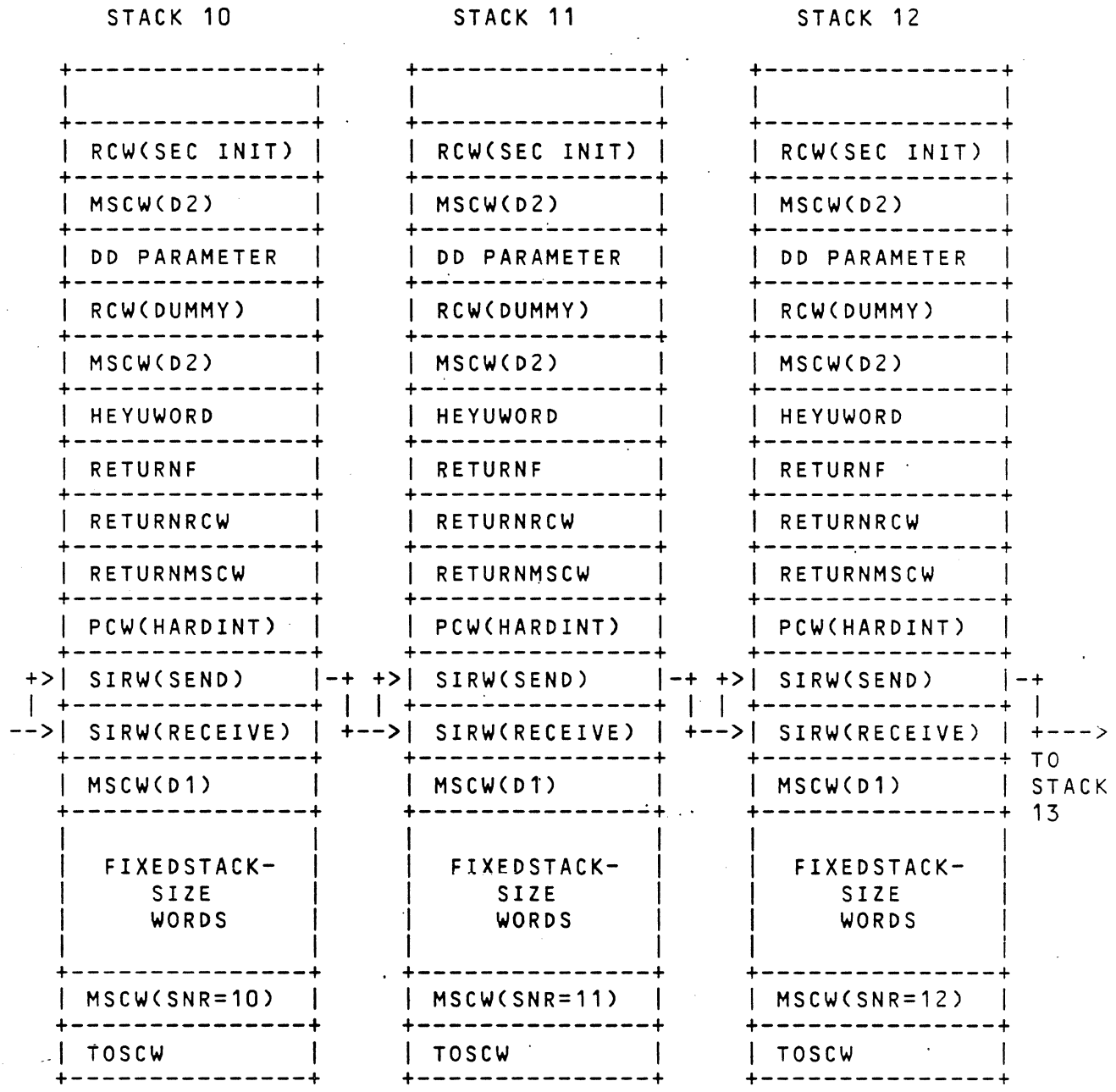
10. The initial PIBDESC is set up. For each processor on the system a PIB is generated and placed in the PIBVECTOR. These structures are allocated in the initialization data area.

11. The BOXINFO array is initialized. This array has information about each box on the system.

12. The slave processors are told to invoke BOOTIT (a define).

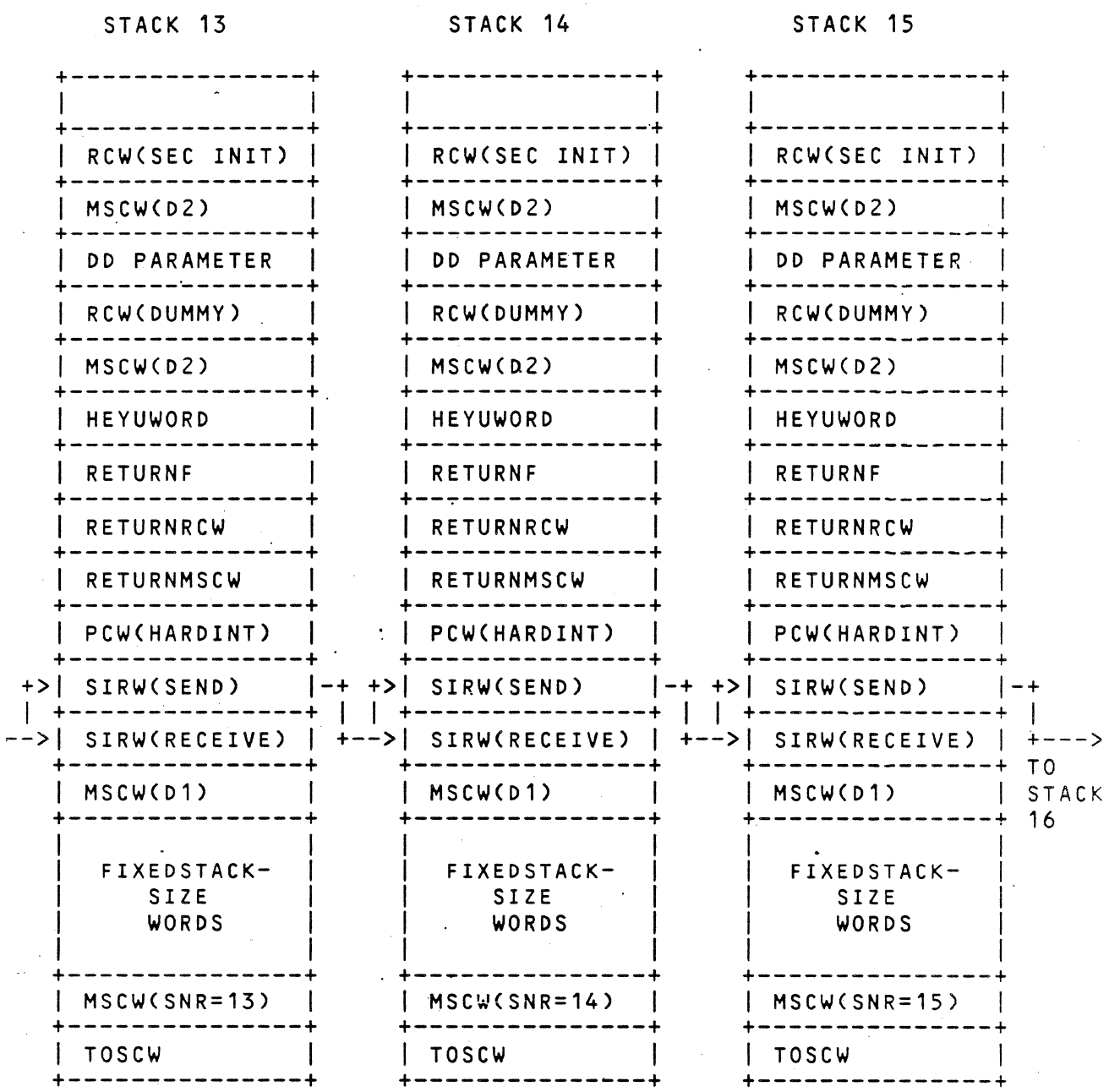
13. The H/L processor calls a procedure MOVETOSECONDARYINITIALIZE which invokes BOOTIT.

    BOOTIT will check to see that no two processors have the same processor number and do a movestack to the proper Halt/Load stack (based on processor number). After the movestack an EXIT is done. The exit will "exit" into SECONDARYINITIALIZE.



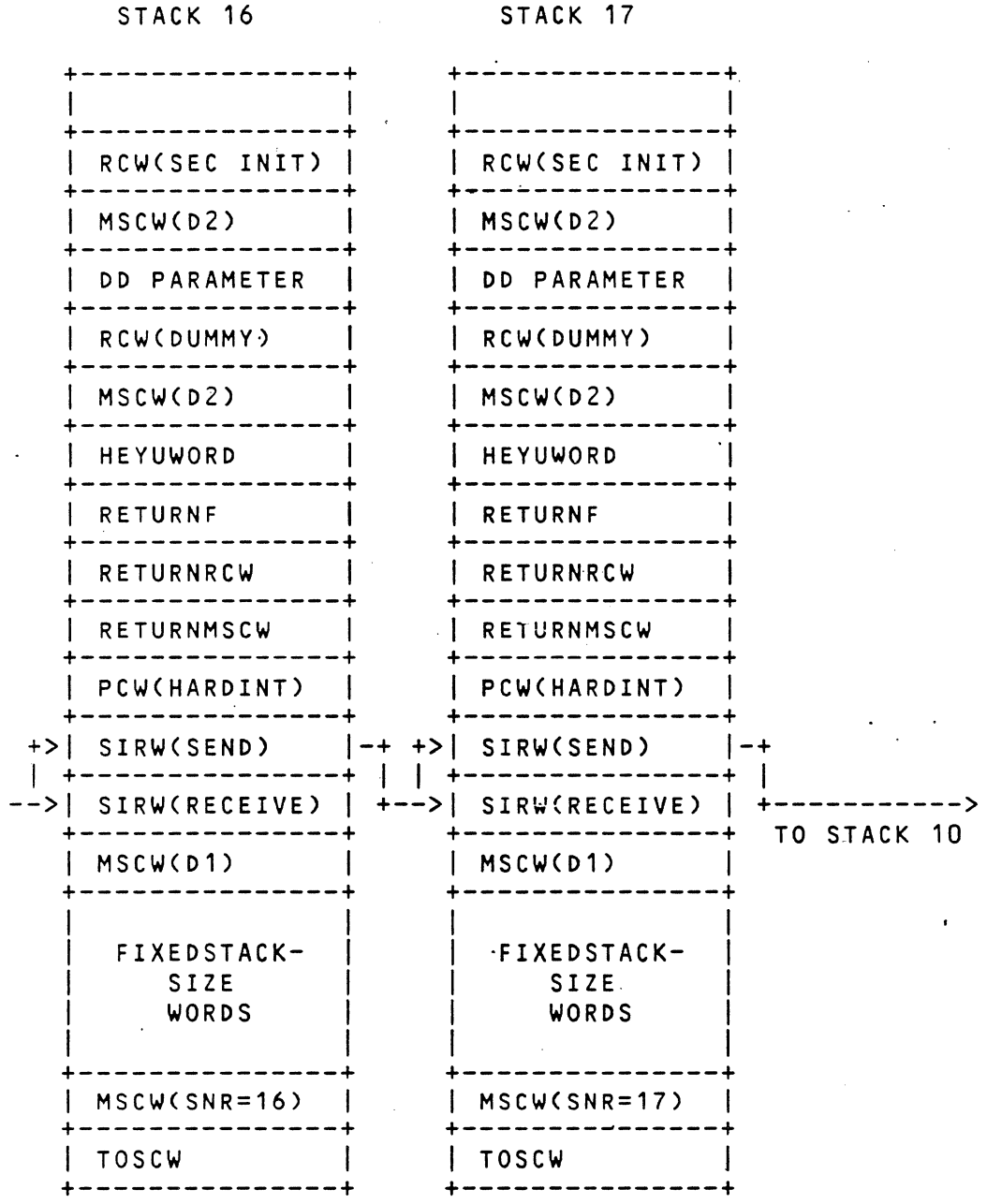
ALL STACKS HAVE THE TOSCW SET UP WITH F POINTING TO THE D2 MSCW AND S POINTING TO THE RCW FOR SECONDARYINITIALIZE.

Figure 3-5. HALT LOAD STACKS (1 of 3)



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

Figure 3-5. HALT LOAD STACKS (2 of 3)



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

Figure 3-5. HALT LOAD STACKS (3 of 3)

OUTLINE OF SECONDARYINITIALIZE

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

SECONDARYINITIALIZE is broken up into two parts. The first part is called SLAVEQUARTERS and is where the slave processors will stay during initialization. The Halt/Load processor will never execute this code. The second part of the procedure is simply referred to as the master's code and it is here that most of the work is done. It is important to note that there is no master slave relationship among processors once the system is completely initialized.

A slave will go into a listen loop waiting for the Halt/Load processor to send him something thru the SEND word. He will pick up the command, do what he is told and send it on to the next processor. Because there is an SIRW chain, this will work for any number of processors.

The following outline is for the Halt/Load processor.

1. Set up word at D0+3 to point to the hardware interrupt routine in SECONDARYINITIALIZE.

2. All DCP's are halted.

3. Global Memory is verified by each processor. Each MOD of Global except the MODs used in initialization will be tested. First the Halt/Load processor does the test. The other processors are told to test the MOD. If any processor has a problem, that MOD will not be used. A MOD is tested as follows.

   a. Call SAVERETURN which will save current RCW and return false.

   b. Try to test MOD (the entire MOD is written and read).

   c. If the MOD is bad HARDWAREINTERRUPT would have been entered. HARDWAREINTERRUPT will use the RCW saved above and return a true. It will look like SAVERETURN returned true. The MOD will be marked offline.

4. Local memory is verified by the processor that can see it (each processor tests its own local memory). Local memory is verified like Global memory.

5. Global memory is linked and MEMLOCK is initialized by the Halt/Load processor.

6. Local memory for each box is linked and MEMLOCK is initialized by the processor that is running in the box.

7. The PROCINFO array is built. This array contains processor times (idle, GEORGE, Pbit) indexed by processor number.

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

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

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

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. These arrays are based on the size of the STACKVECTOR (one entry per stack). The STACKSTATUS array contains priority and READYQ linkage. The STACKINFO array contains NEEDAROW bit, stack visibility, stack kind and other information.

    The READYQ is initialized to point to a null entry (the MCP stack).

13. The PIBVECTOR is established. The size is based on the size of the STACKVECTOR.

14. The EXTPIBDESC is set up which is used by BNA for tasks running on another system.

15. The STACKVECTOR is established.

16. The H/L stacks are moved to the STACKVECTOR.

17. Pseudo stacks are placed in the STACKVECTOR. These stacks point to 64K blocks of memory and allow an SIRW to point to any word in memory.

18. The MCP's PIB is set up.

19. GETAREA/FORGETAREA structures are established.

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

    a. INTERLOOPERs RCW is set up.

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

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

    d. Normal HARDWAREINTERRUPT routine is set up.

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

22. The peripherals are initialized by PERIPHERALINITIALIZE. This includes setting up fail IOCBs, testing channels and building ringwalk information. In addition, the DISKFILEHEADERS stack is built.

23. The memory dump to tape PCW is put in the DO stack.

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

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

26. It is now time to get the processors into true stacks.

    If this is a monolithic system, the slaves are told to get into another stack. They do this by calling GEORGE who will find them another stack (an idler). A procedure called RUNONE is called to initiate a stack for ETERNALIR. Thus, the slaves will begin running an idler and a copy of ETERNALIR has been initiated for the Halt/Load processor.

    If the system is tightly coupled a copy of ETERNALIR is initiated (by RUNONE) for the global box. A copy of ETERNALIR is initiated (by RUNONE) for each local box by the processor in the box. The slaves move to their copy of ETERNALIR.

27. The alternate stacks are forked. The DCALGOL queue stack is set up and the MCP overlay file is initialized.

28. The Halt/Load processor moves to its copy of ETERNALIR. Thus, the processors will execute ETERNALIR or an idler.

INITIALIZATION PART OF ETERNALIR

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

1. Return the Halt/Load stack numbers.

2. Release initialization memory areas (code and data).

3. Fork ANABOLISM.

4. Call ANABOLISM to start the forked independent runners.

ETERNALIR for a non-Halt/Load processor will do no other

initialization.

Thus, for a tightly coupled system there is one ETERNALIR per box and one ETERNALIR in global. In a Monolithic system there is one ETERNALIR in the system.

At this point the system is interrupt driven.

OUTLINE OF STARTSYSTEM

1. Wait for CONTROLLER to start running so the operator can see ODT messages.

2. 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.

3. Call VERIFYFAMILY to read in the header for SYSTEMDIRECTORY. This is the header that describes the FLAT DIRECTORY for the family. This call on VERIFYFAMILY will also result on a call on DISKMAPPER. DISKMAPPER will set up all available disk tables for the H/L family and will complement the family's FLAT DIRECTORY.

4. 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.

5. If SYSTEM/ACCESS does not exist, call INITIALIZECATALOG to build a header for the ACCESS STRUCTURE. INITIALIZECATALOG will also get the first row of the structure and initialize the PAST portion of it.

6. Remove all off-line units from the PAST. Verify all other family's.

7. Set up USERDATAFILE (USERCODE file).

8. Let CONTROLLER know that the directory can be used.

9. Restore working set factors and fork WSSHERRIFF if OLAYGOAL is greater than 0.

10. Start supervisor program.

11. Set up idle display for lights based on machine type.

SECTION 4

DISK MANAGEMENT

INTRODUCTION

Disk 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 are of primary interest.

HALT/LOAD DISK LAYOUT

Figure 4-1 is a representation of the halt/load unit. What is called the LABEL BLOCK in this manual is actually the first 23 segments on the halt/load disk unit. The LABEL BLOCK 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 4-1 was created by the SYSTEM/LOADER.

When the loader is run for the purpose of cold starting, it has two important tasks. The first is to load the MCP from tape to disk and the second is to set up the FLAT DIRECTORY and LABEL AREA.

Figure 4-2 shows the layout of the halt/load disk as set up by the SYSTEM/LOADER. In absolute segment 0 is the bootstrap, followed by the label in segment 4. Most of the first 23 segments are not used but are reserved. The FLAT DIRECTORY header is pointed to by a word in the pack label.

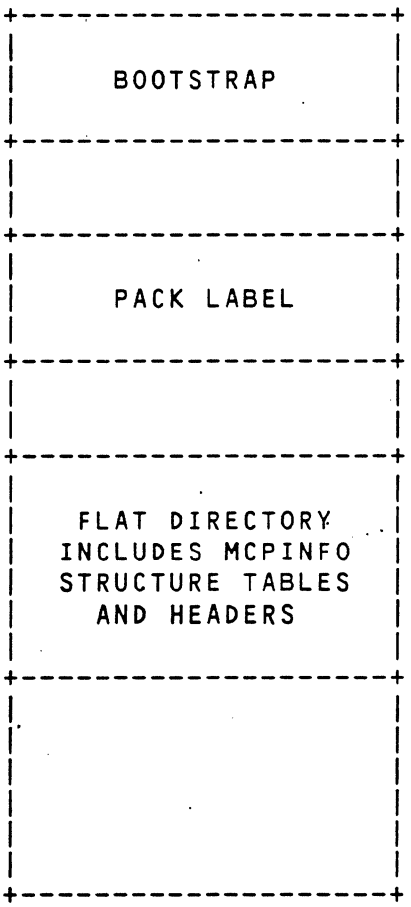


Figure 4-1. Disk Initialization

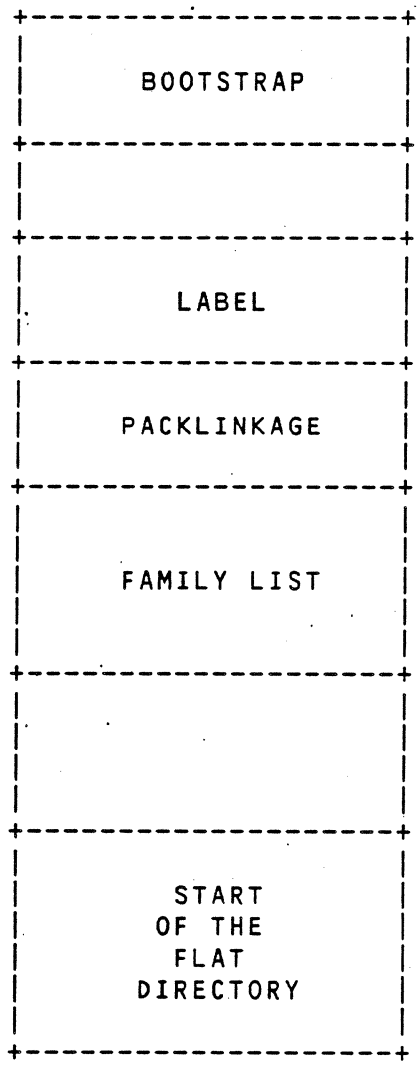


Figure 4-2. Halt/Load Disk Layout

TERMINOLOGY

We consider a disk unit to be each part of the system which is a contiguous area of disk for addressing purposes. For removable and non-removable disk it will be a disk pack spindle.

File location (unit or units where a file resides) is specified with a FAMILY NAME. A FAMILY NAME is a unit or group of units which has its own directory stored on one unit of the group. Thus, when accessing a file, the file name and family name must be given. For example, FILE ON PAKFAM1 or 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 mount 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 a HEADER-NAME BLOCK 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 family (see the DD ODT command). Each copy of the FLAT DIRECTORY is updated simultaneously. The name of the FLAT DIRECTORY is SYSTEMDIRECTORY/nnn (nnn is the FAMILYINDEX where the directory resides).

The FLAT DIRECTORY has 600-segment rows. The first row of the FLAT DIRECTORY (figure 4-3) contains a segment zero, audit

area, MCP structure tables, FLAT DIRECTORY pointers, B7700 peripheral table and HEADER-NAME BLOCKS.

Segment zero contains information about the row of the FLAT DIRECTORY and its relationship with other rows.

The audit area is used when the directory is being updated to enable the system to recover should the system fail during the update.

The MCP structure tables contain pointers (record numbers) to MCPINFO, swapper parameters, SL functions, configuration, peripheral table (B7000) and UNITADDL. These tables are pointed to by segment zero of the FLAT DIRECTORY. Of particular importance is the MCPINFO table which is created by SYSTEM/LOADER (or a CM) and maintained by the MCP. Its basic function is to hold information that should be remembered across halt/loads. Information contained in this table includes the last mix number used, name of the intrinsics file, number of auto printers going and other run time information. The UNITADDL table contains information about peripheral units that must be retained across a halt/load.

Another part of the FLAT DIRECTORY contains headers for the FLAT DIRECTORY (sometimes referred to as a self-pointer). The table referred to as B7700TABLES is where the parameter card information concerning peripherals and DCP's is kept.

The remainder of the first row contains HEADER-NAME BLOCKS. All other rows in the FLAT DIRECTORY contain headers with only one segment reserved for MCP use.

A HEADER-NAME BLOCK (figure 4-4) in the FLAT DIRECTORY contains information about the file. A HEADER-NAME block is called a DISK header in the MCP. The actual header layout is contained in the DISKFILEMANAGER module of the MCP. The general types of items found in a disk header are header size, opencount, filekind, security information, block size, record size, units (words, characters), timestamps, row size and end of file pointers (segments and number of valid bits in the last segment). The disk header also contains row address words which contain family index (row not fired up) or unit number (row fired up) of the row. The row address word also contains the base address of the row and DMSII row lock out bits (read and write). The row address words are followed by the name in standard form.

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. This information points to the header in the FLAT DIRECTORY.

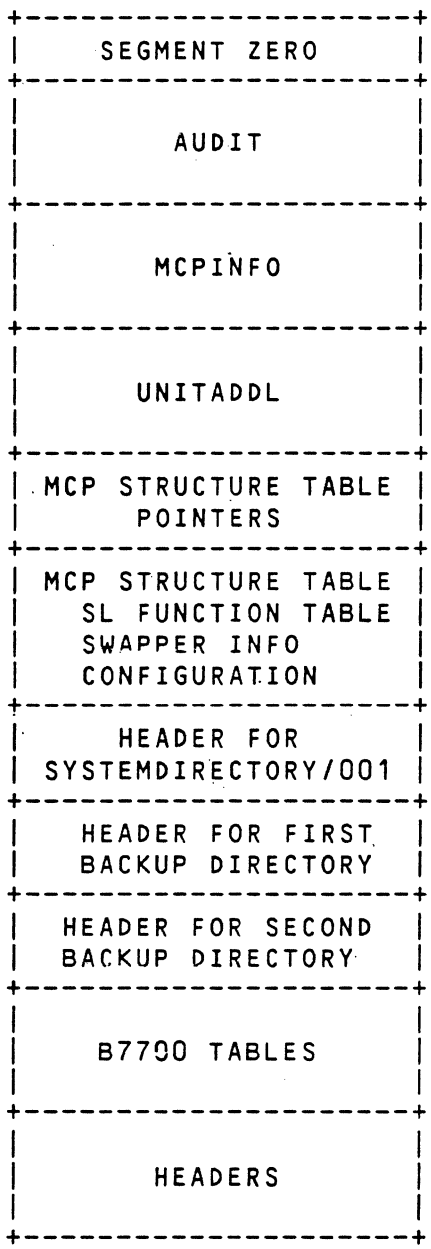
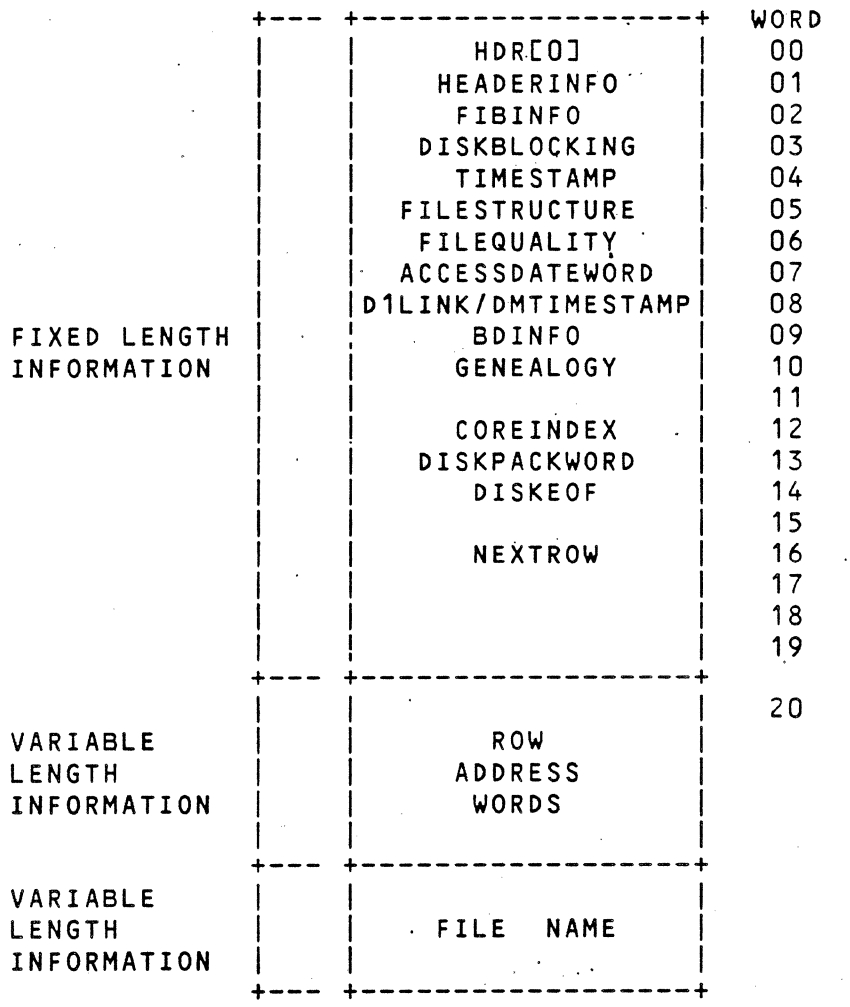


Figure 4-3. First Row of the Flat Directory



ROW ADDRESS WORDS POINT TO ROWS OF FILE NAME POINTED TO BY HEADERINFO WORD

Figure 4-4. Header-name Block

ACCESS STRUCTURE

The ACCESS STRUCTURE is used to locate a header in the FLAT DIRECTORY. There is one ACCESS STRUCTURE for the system. Up to two copies (three total) of the ACCESS STRUCTURE can be maintained on other members of the family (see the AD ODT command). Each copy of the ACCESS STRUCTURE is updated simultaneously. Given a family name and file name the ACCESS STRUCTURE is used to produce the position of the HEADER-NAME BLOCK in the FLAT DIRECTORY (see figure 4-5).

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. If no DL is specified the ACCESS STRUCTURE 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. The ACCESS STRUCTURE is structured in 8-segment blocks.

The ACCESS STRUCTURE is in two parts (see figure 4-6), 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. The FAST blocks 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".

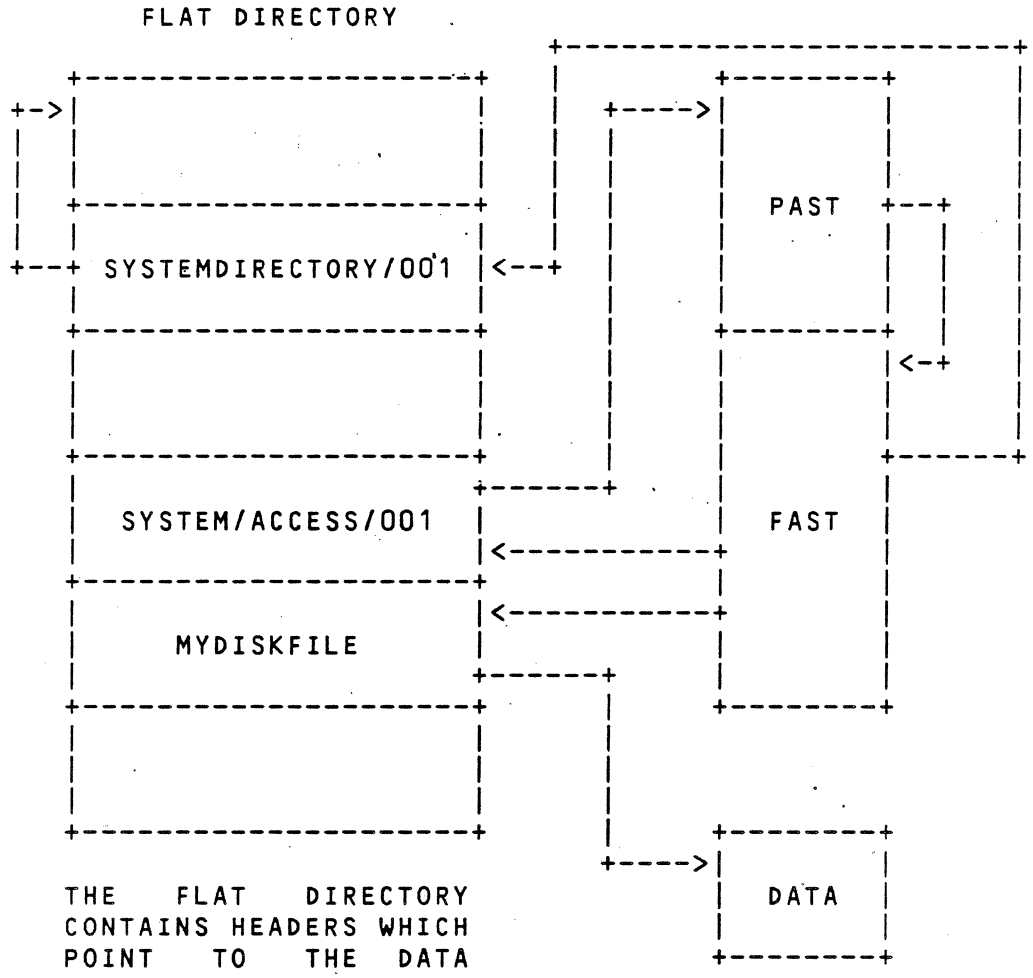


Figure 4-5. Disk Files

SYSTEM/ACCESS/001

FIRST ROW OF ACCESS STRUCTURE

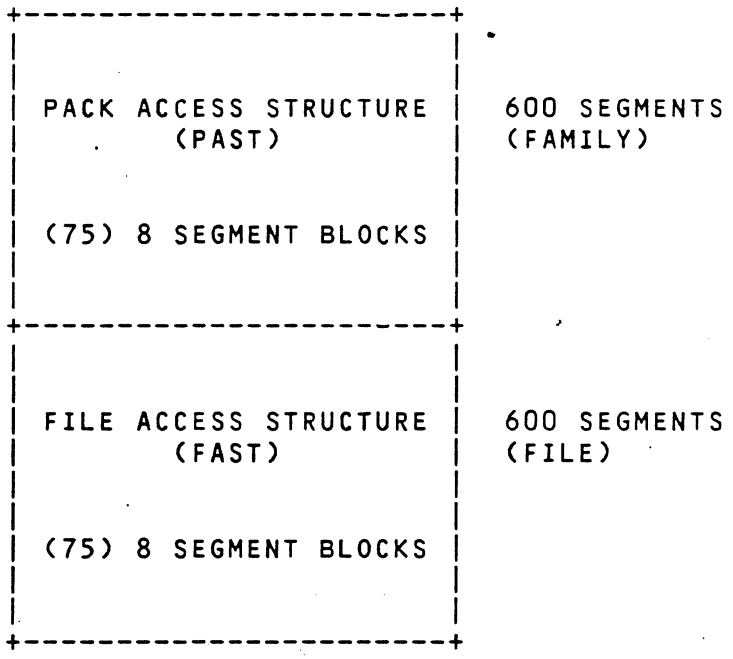


Figure 4-6. Access Structure

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, a block is calculated based on an arithmetic manipulation of the characters in the family name (ie. HASHED). A simple search is then performed on the PAST block until the entry with the given family name is found. This entry contains pointers identifying the section of the FAST which is applicable to the family.

There are two fields in the PAST entry. The first is the starting FAST block for the family's NON-USERCODE files. The second field is the starting FAST block for the family's USERCODE files. This is the initial set up for all families and represents the beginning of two separate strings of blocks. One string for NON-USERCODE files and the other string for USERCODE files.

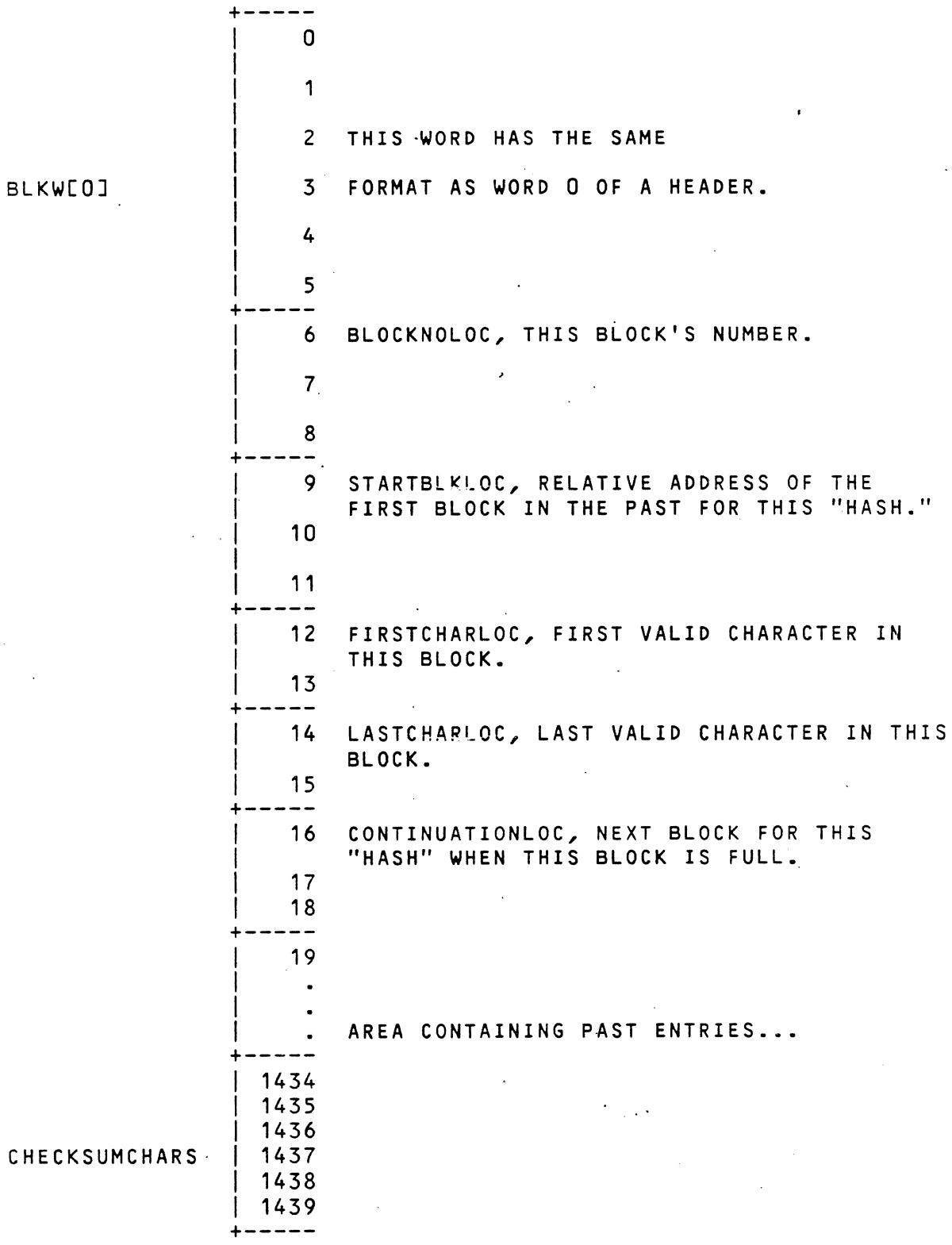


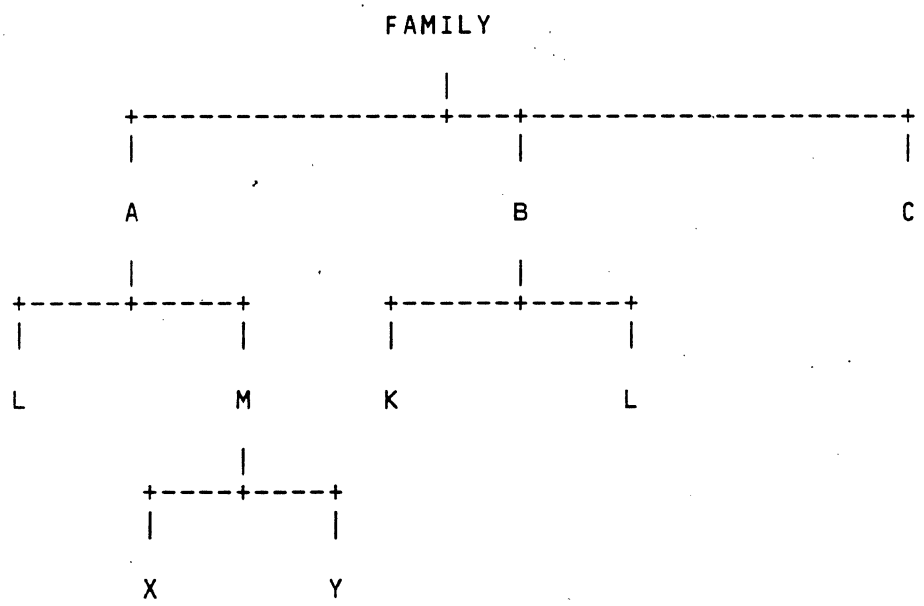
Figure 4-7. Pack Access Structure (PAST) Block

When a family is brought on line, the MCP procedure READADISCLBL is forked. This procedure gets a memory area for an information array 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, all information from the PAST entry 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 4-7.

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:



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. Brothers are stored contiguously and 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). Entries are termed 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 (pointer to the first son).

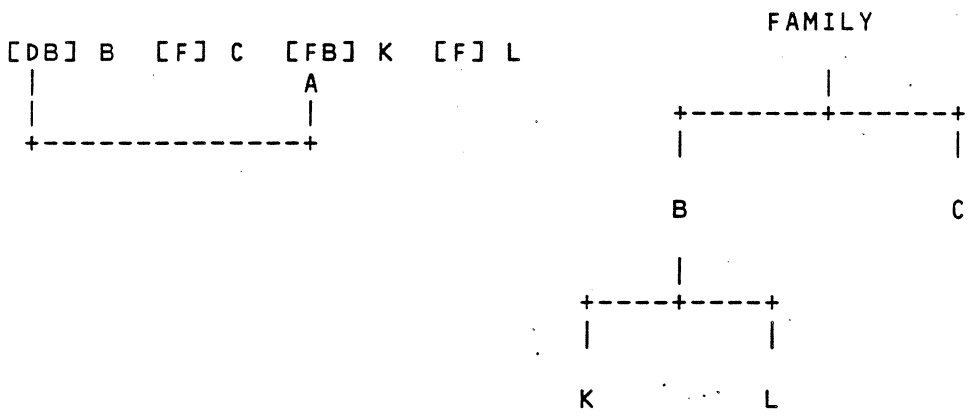
The following notation is used to demonstrate the tree structure.

F. This identifier is a file.

D. This identifier is a directory.

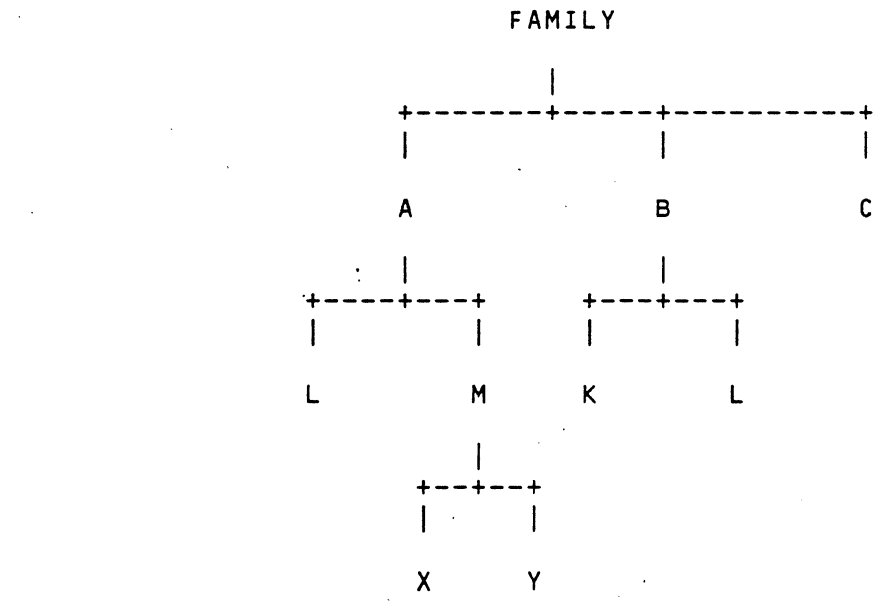
B. This identifier has a brother (the next entry).

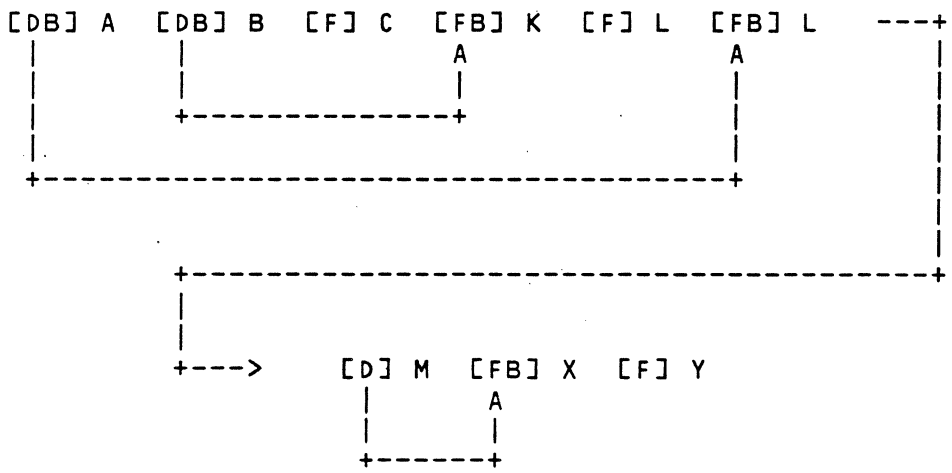
Suppose that the family described above started with the files B/L B/K and C. The physical layout of the FAST block for these entries would be:



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 will be as shown on the next page.


FAMILY
A
B
C
L
M
K
L
X
Y


[DB] A
[DB] B
[F] C
[FB] K
[F] L
[FB] L
A
A
[D] M
[FB] X
[F] Y
A

An entry in the FAST composed of several characters. The first character specifies the length of the entry in characters. In addition, it contains a BROTHERF (this file has a brother), a DIR bit (this entry points to a directory entry) and a FYLE bit (this entry points to a file). If both the DIR bit and the FYLE bit are off the last two characters in the entry contains a pointer to a continuation block where the entry may be found.

The next character is the length of the name entry in characters.

The next N characters are the name. N represents the length of the name.

If the FYLE bit is on, the four characters following the name point to the header in the FLAT DIRECTORY. These characters specify the length of the header in words and the address of the header in the FLAT DIRECTORY. The address is relative to the beginning of the FLAT DIRECTORY.

If both the FYLE bit and the DIR bit were on, then the four characters described immediately above would have been followed by two characters. These characters would point to the next level and are 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 four characters described above would have not existed in the entry.

Figure 4-8 shows how an entire FAST block is layed out. Notice its similarity to the PAST block layout of figure 4-7.

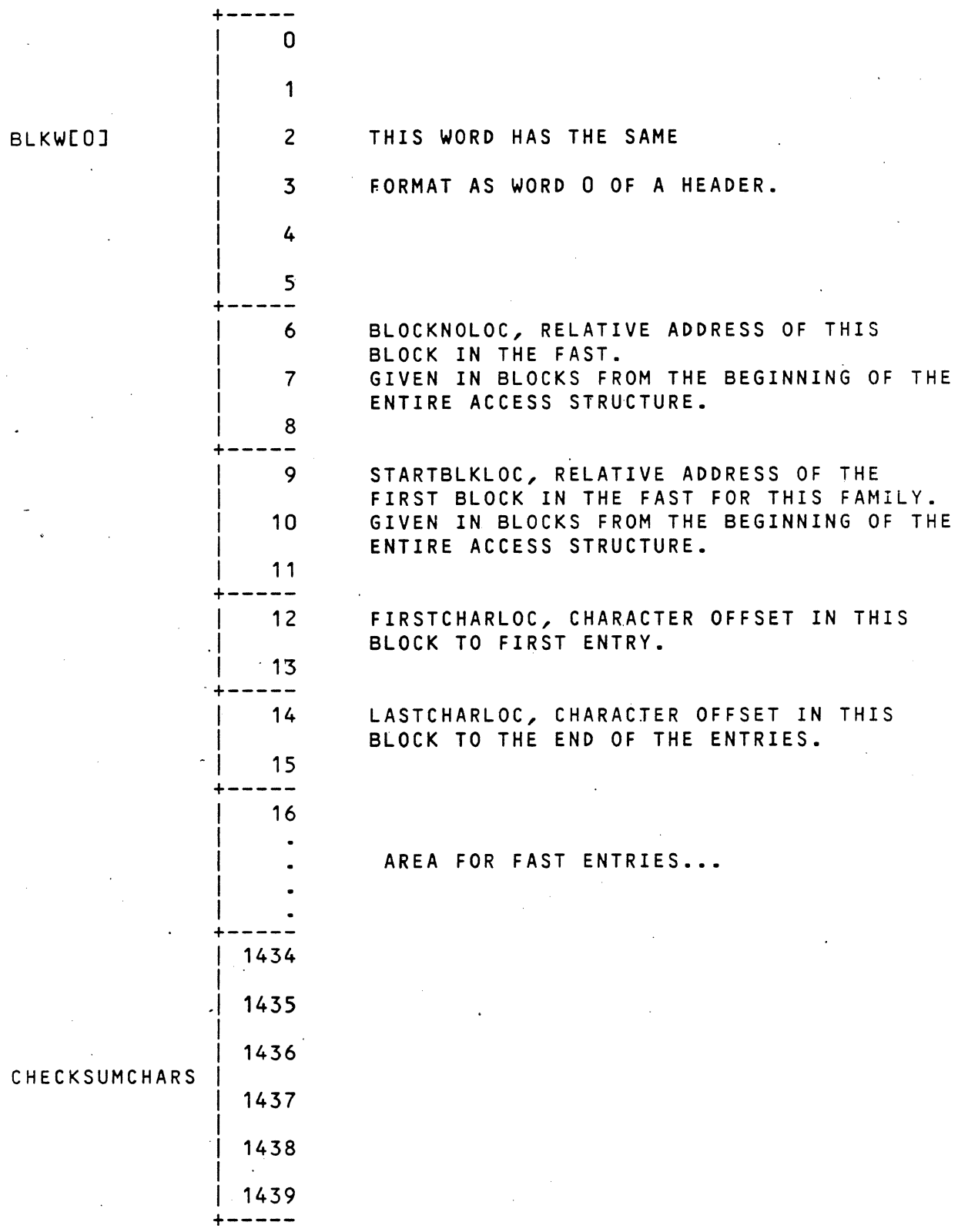


Figure 4-8. File Access Structure (FAST)

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, allocated to a file that has an entry in the disk directory (termed a permanent file) or temporarily in use. A temporary file is a file which has been opened but not entered in the 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. That is, 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.

Directory complementing is done from the FLAT DIRECTORIES present on the system at restart time. This is a very fast process, since the large blocking factor of the FLAT DIRECTORIES enables them to be read with minimal I/O operations.

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. It 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. 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 disk available table. 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 4-9. 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 4-9 but have been redrawn in figure 4-10 to better display them.

The CHUNKLISTS (fine tables) of figure 4-10 are the last link to available user disk locations. All CHUNKLIST words contain two fields. The DISKAREASIZEF is the number of segments in the area and the DISKAREAADDRF is the absolute address of the area.

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 FAMILYINDEX. With the FAMILYINDEX, the FMLYLIST (figure 4-9) 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:

USERDISKINDEXF. This is an index into the array USERDISKLIST of figure 4-10 and is used to relate the word GETHEAD or FORGETHEAD with a particular CHUNKLIST.

EUNOF. This field contains the family index of the member who owns the CHUNKLIST. Two members should never be pointing to the same CHUNKLIST.

SEGADDRESSF. This field contains different information 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 USERDISKINDEXF. In GETHEAD this field contains the size in segments of the largest area in the CHUNKLIST pointed to by USERDISKINDEXF.

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.

When GETUSERDISK finds an area, 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).

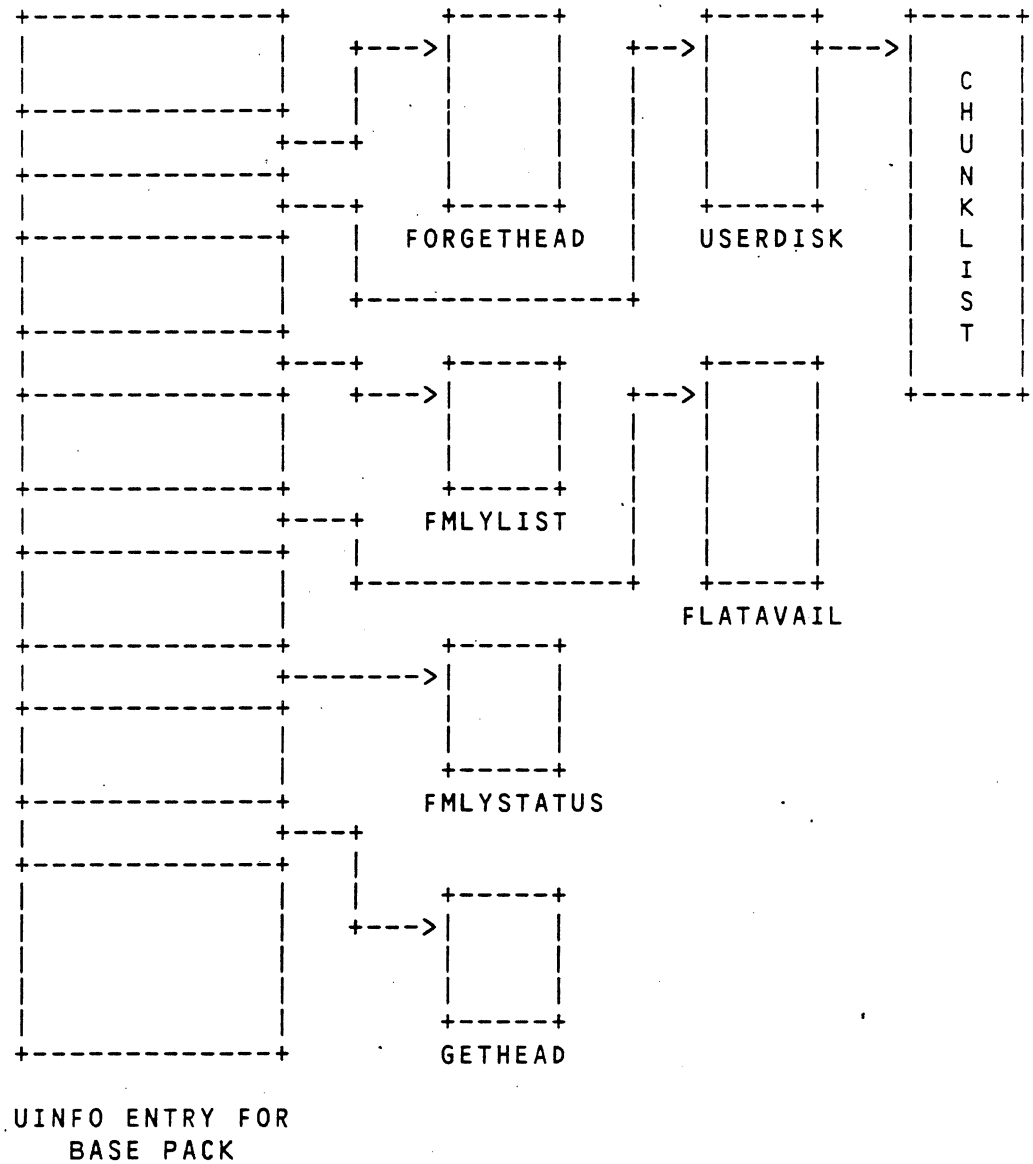


Figure 4-9. Disk Information Arrays

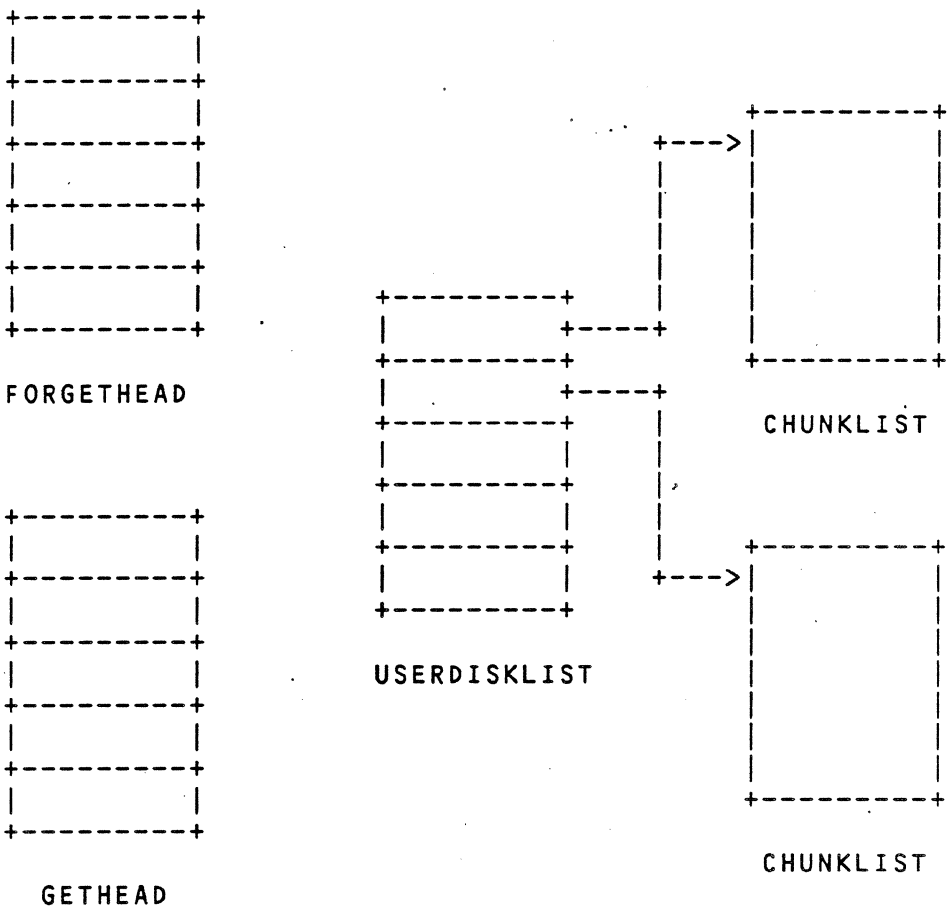


Figure 4-10. Available Disk List

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. These structures are FMLYLIST and FMLYSTATUS.

These two arrays are shown in figure 4-9 and will be explained in the following paragraphs.

FMLYLIST

FMLYLIST, called PACKINFO on memory dumps, contains information about the family members (one word per member) and is indexed by FAMILYINDEX. 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. All other words in FMLYLIST will contain the following after DISKMAPPER is through with the array:

PBITF	If family member present.
DKBACKUPF	If contains BACKUP DIRECTORY.
DKGOINGFLAT	If getting a FLAT DIRECTORY.
IADPKF	If IAD pack.
DKEUNBRF	EU number of family member.
DKSPEEDF	Speed attribute (0 for pack).
DKTABLEINDEXF	Index into DISKTABLE (MCP table).
DKAREACLASSF	AREACLASS attribute (disk class).
DKSERIALNBRF	Members serial number in binary.

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 information indicating which storage units/packs on a family member are either not ready or in a write lock-out condition. Within each word the DKCLASSCNTF indicates the number of classed segments and DKSWITCHESF indicates if an SU/PK is locked out.

FLATREADER

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 available 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.

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). 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.

FLATREADER calls the global procedure TAKEBACKDISC to make the entries in the DKTABLE and it is TAKEBACKDISC that should be referred to if more information is desired on this table.

DISKMAPPER constructs the DISCAVAIL tables shown in figure 4-10 from information in the DKTABLE that was previously passed to FLATREADER.

FLATAVAIL TABLE

FLATAVAIL is the array containing available FLAT DIRECTORY header locations. This array is actually composed of an address list, size list and miscellaneous part.

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. The AVAIL word in the miscellaneous part points to the next available location in the list.

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 ANABOLISM and ETERNALIR. The special stacks discussed in this section are the DISKFILEHEADERS stack, INTRINSICS stack and the DATACOM QUEUE stack. This section is followed by some procedure outlines.

INDEPENDENT RUNNERS

Procedures in the MCP may be FORKED, which is analogous to an ALGOL RUN. When ALGOL procedures are RUN, the MCP creates a separate stack for the procedure (an external program) 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 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 0

IRSTACKSIZEB is defined as 380.

IRPRIORE is defined as 101.
```

INSTRUCTION	COMMENT
MKST	Start of the fork statement's code.
NAMC (0,XXX)	ETERNALIR's PCW.
	If the procedure being forked was passed parameters, they would be located here.
MKST	Prepare to enter FORKHANDLER.
NAMC (0,XXX)	FORKHANDLER's PCW.
ZERO	Box number.
LT16 17C	Size of the process stack.
LT8 65	Priority of this independent runner.
LT48 09C5E3C5D9D5	First part of the name of the IR.
LT48 C1D3C9D90000	Second part of name of the IR.
JOIN	Make the two name words double precision.
ENTR	Enter the FORKHANDLER procedure.
ENTR	When FORKHANDLER is executed, the IRW pointing to ETERNALIR's PCW will be replaced with an IRW pointing to JUSTEXIT's PCW. Thus, this ENTR will enter JUSTEXIT. JUSTEXIT will simply exit back into the procedure that invoked the FORK.

The STACKSIZE parameter (second parameter) provides the stacksize for the Independent Runner. In addition, if bit [47:1] is set the Independent Runner will be a control program. If bit [46:1] is set the Independent Runner is a one only Independent Runner.

The PRIORITY parameter (third parameter) provides the priority for the Independent Runner. In addition, If bit [46:1] is set the Independent Runner will be visible on the ODT.

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?

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 ANABOLEVENT.

FORKHANDLER is the procedure that inserts requests into the queue. This queue is protected by a lock called FROCK. 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.

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, STACKSIZE, ENTRYPOINT, and NAME (MYNAME). If there are parameters, they are left in the area. A word in the PIB (TASKPARAMS) 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 gets space for and initializes a process stack and, places the stack in the READYQ.

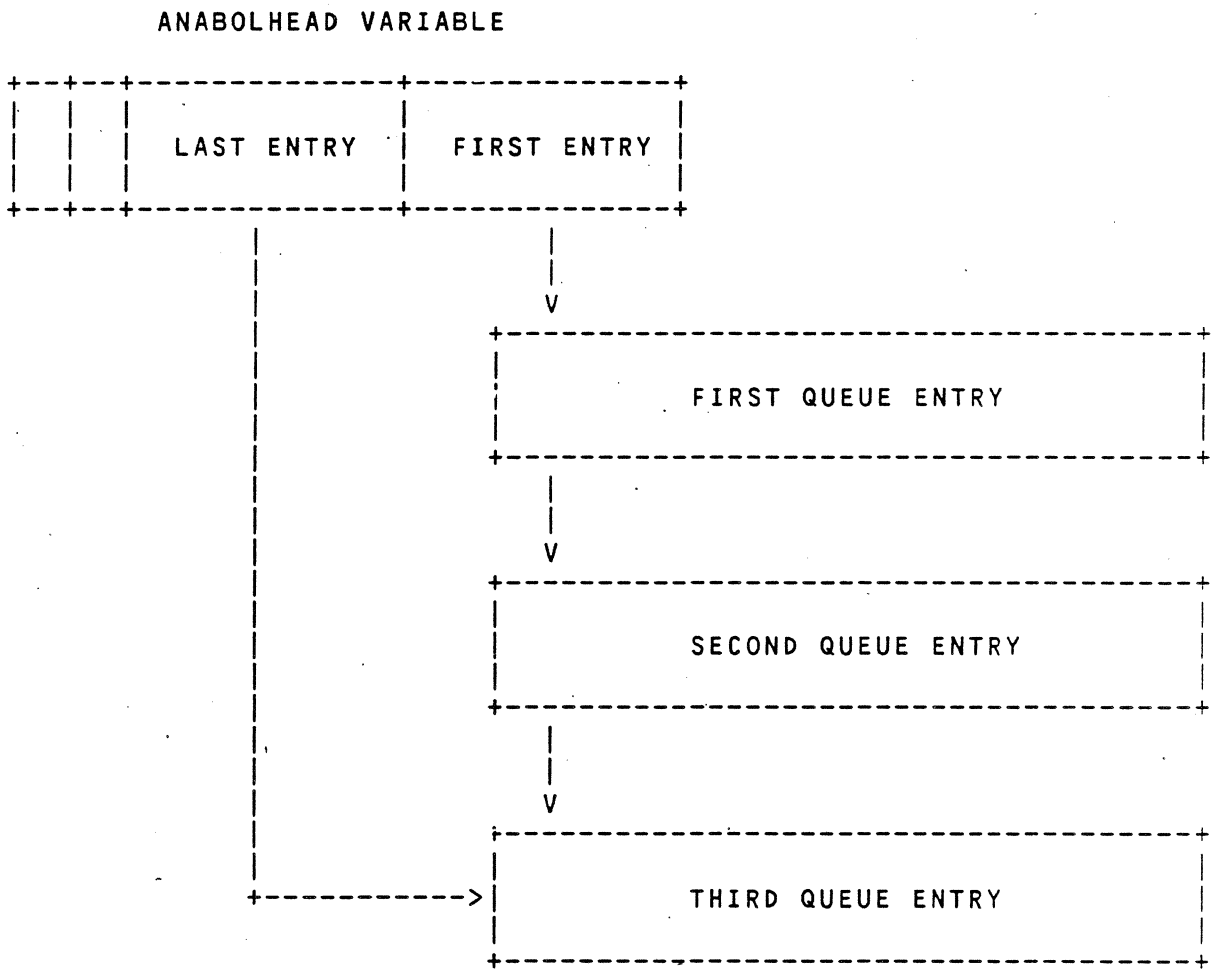


Figure 5-1. FORK Queue

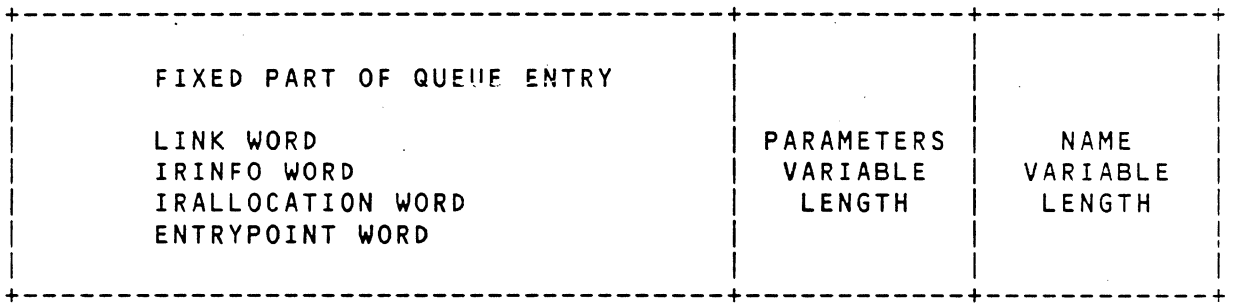


Figure 5-2. Queue Entry

ETERNALIR ROUTINE

ETERNALIR is forked by SECONDARYINITIALIZE during system initialization. A copy of ETERNALIR is FORKED for each box on the system. That is, one copy for each local box (memory component) and one copy for GLOBAL. A monolithic system will only have one copy of ETERNALIR.

ETERNALIR can act as several different independent runners. Its purpose is to always be around when one of these jobs is to be performed, thus not requiring a separate independent runner to be in memory for these relatively small tasks.

When one of the other MCP procedures wants ETERNALIR to do a queue driven function it calls RILANRETE passing three parameters which are subsequently linked into a list. RILANRETE then causes ETERNALEVENT to wake up the proper ETERNALIR.

RILANRETE PROCEDURE

RILANRETE is called when a queue driven request is made to ETERNALIR. 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. This queue is protected by a lock ETERNALIRQLOCK. It will then CAUSE the proper ETERNALIR event.

SPECIAL STACKS

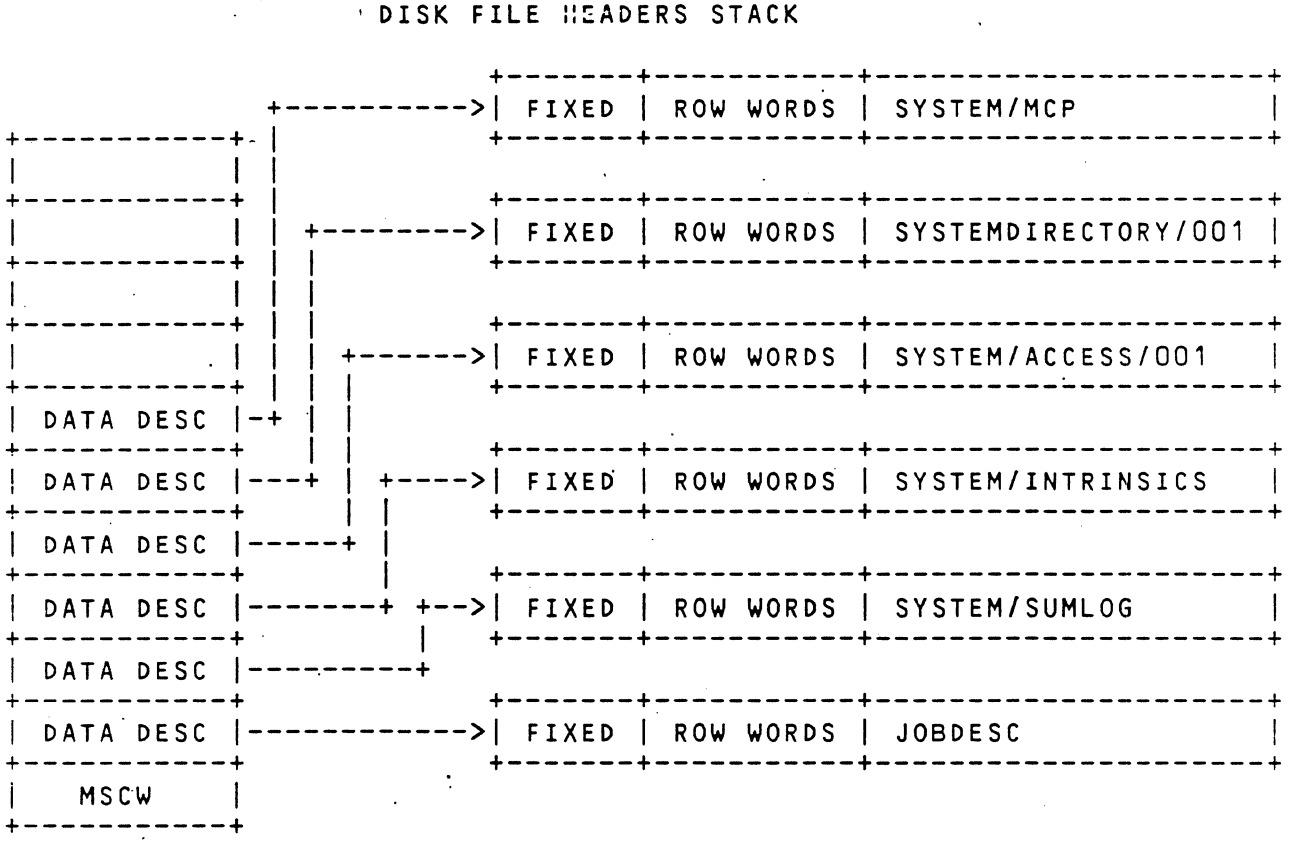
The special stacks to be discussed here are the DISKFILEHEADERS stack, INTRINSICS stack and the DATA COMM QUEUE stack. the INTRINSICS stack, set up by the procedure INITIALIZEINTRINSICSTUFF may not be set up at system initialization time as are the other two but it may be set up later as a response to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

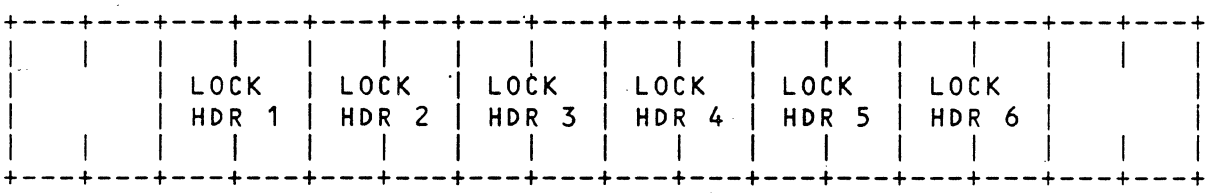
The headers that are nearly always seen in the header stack, as well as the stack itself, are shown in Figure 5-3. Note that the SYSTEMDIRECTORY header is for the FLAT DIRECTORY and that the JOBDESC header is used by CONTROLLER to save the rules for controlling ODT screen activity, job queue information, 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.



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.

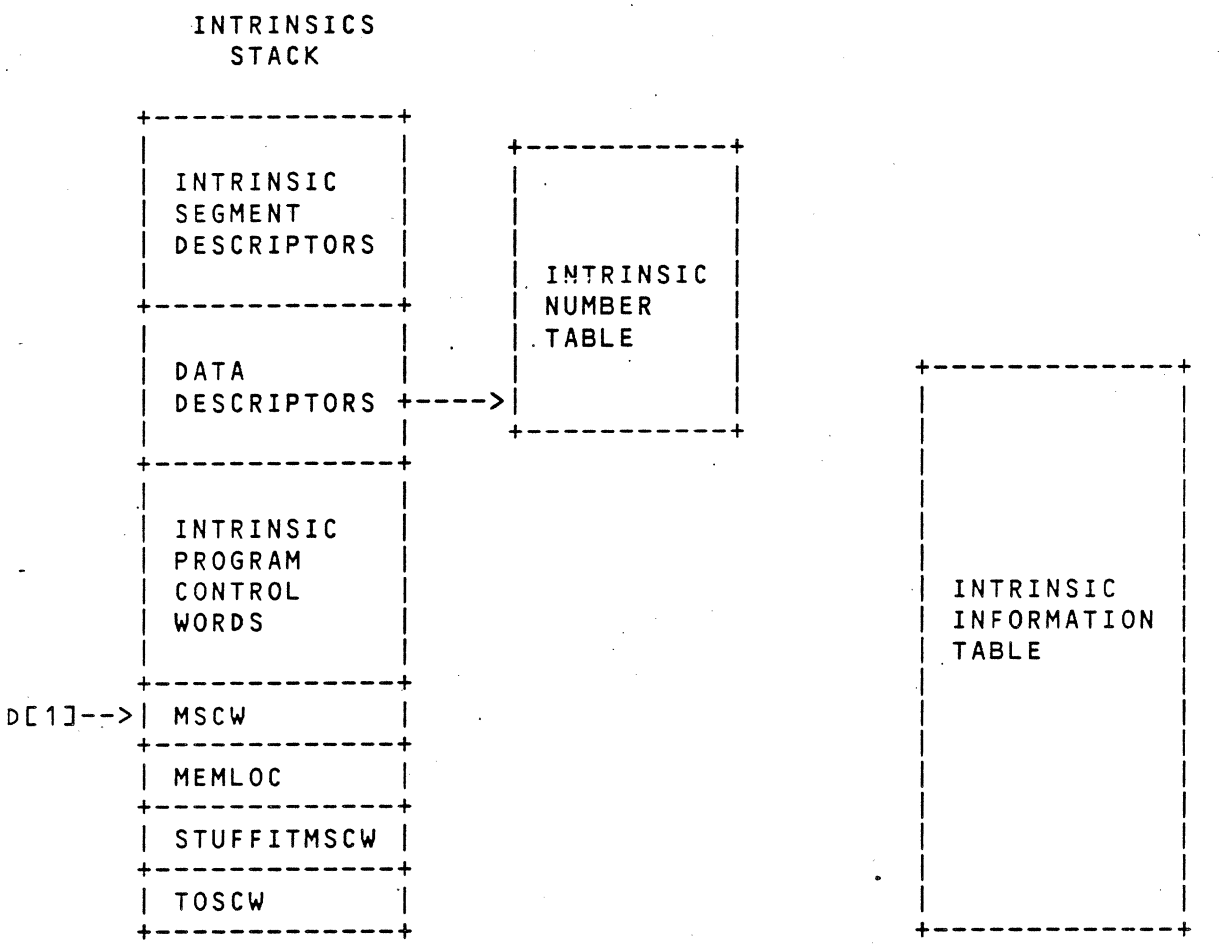
An intrinsic's PCW is located by an identifying INSTALLATION number and INTRINSICS number. These two numbers are found by compilers by use of the INTRINSICS INFORMATION TABLE shown in Figure 5-4. To find an INTRINSIC, the MCP must be supplied with a word containing the INSTALLATION number in bits 23:11 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

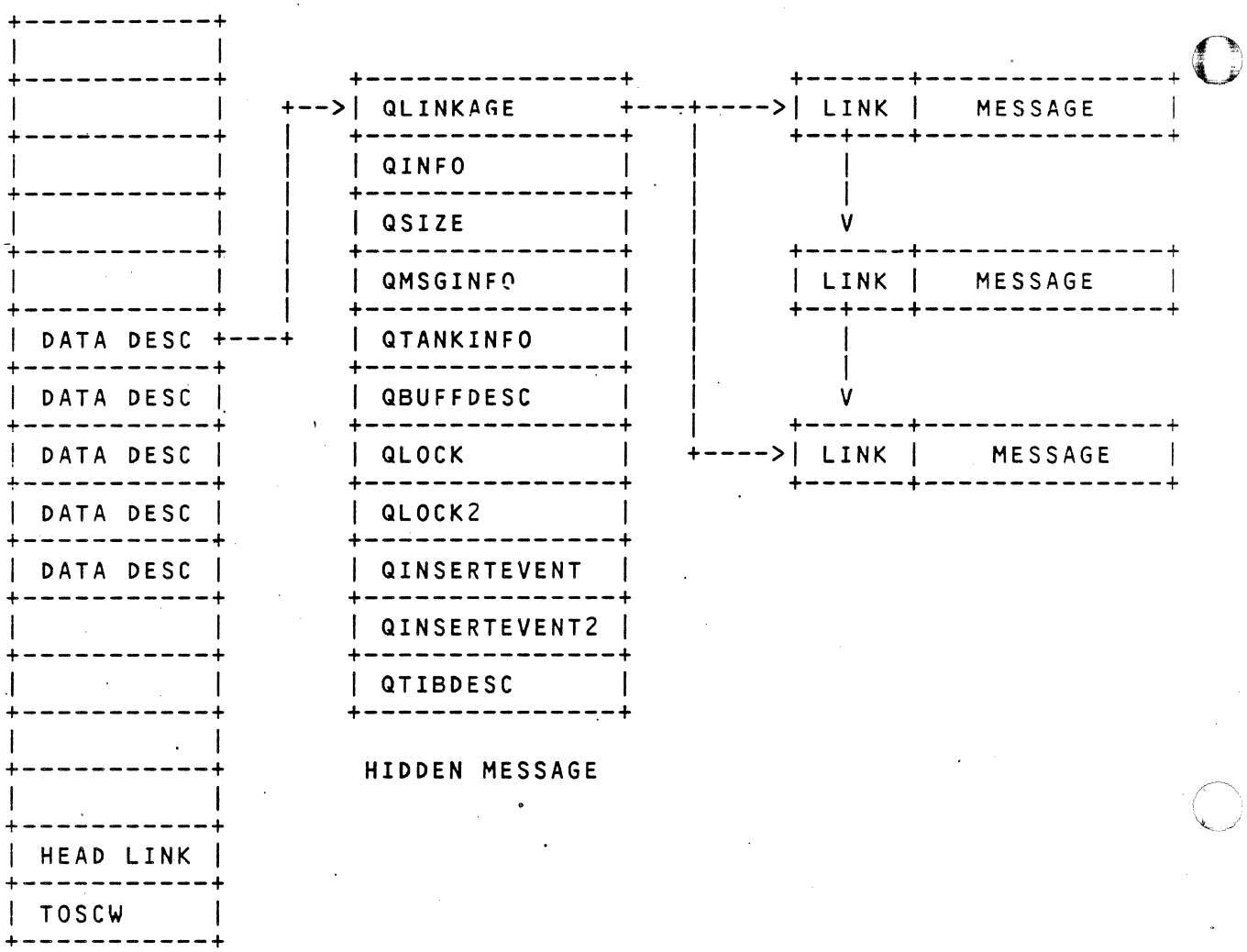
Besides being a central location for queues declared in DCALGOL and FILE INPUT QUEUES for programs with remote files, the DATA COMM QUEUE stack serves as the communication link among Independent Runners. The main IR'S using the queues in this stack are CONTROLLER and CONTROLCARD (WFL).

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.

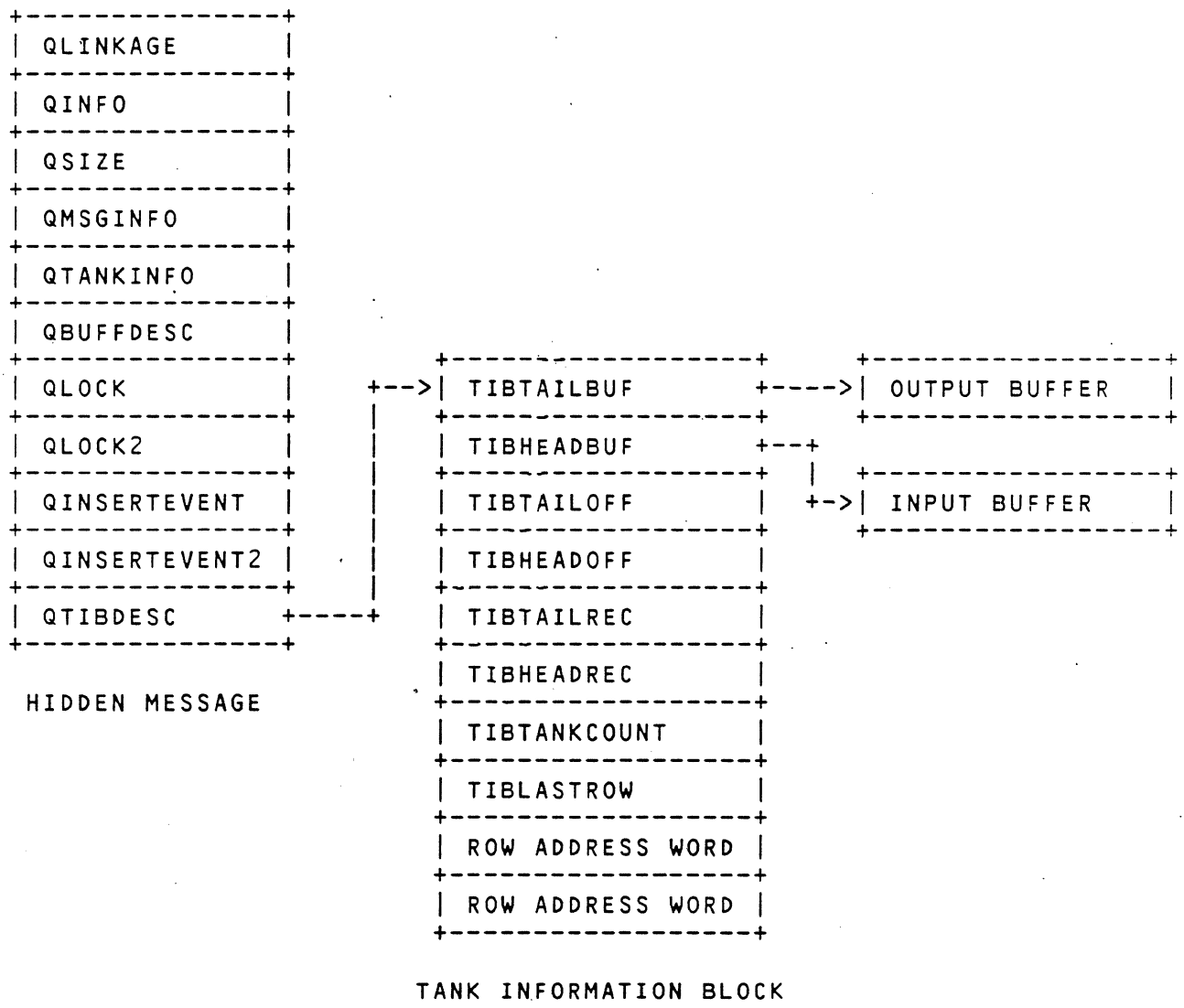
As messages are placed in a queue, as can be seen in figure 5-5, the queue's memory limit may be reached. If the queue's limit is reached, a TANK INFORMATION BLOCK (TIB) is built, disk rows are obtained with the procedures GETUSERDISK and subsequent messages for the queue are TANKED, (i.e., written to 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

OUTLINE OF FORKHANDLER

FORKHANDLER is called to process a FORK statement. This procedure will build a queue entry and place the entry in the queue for ANABOLISM.

1. The address of the IRW which points to the procedure to be forked is computed using the MSCW.

2. If this is a one only independent runner (stack size less than zero) and it is already running, this request will not be done (a branch is made to step 9). If the IR is not running it will be set to a running state. The running status is maintained in an unused bit in the PCW. The bit is checked under control of the lock FROCK.

3. GETAREA is called to get an area to save parameter information. The size of the area is based on number of parameters and size of the name.

4. The fixed IR information is placed in the area.

5. If the IRW to the procedure to be forked is not an SIRW it is stuffed and placed in the area.

6. If there are parameters they are moved from the stack to the area.

7. The name is moved into the area.

8. The area is linked into the queue for ANABOLISM. This queue is protected by the lock FROCK. The event ANABOLISM waits on is caused (ANABOLEVENT).

9. The IRW in the stack is changed to an SIRW to JUSTEXIT. FORKHANDLER will exit. The callers code will do an ENTR, which will enter JUSTEXIT, which will exit.

OUTLINE OF ANABOLISM

ANABOLISM is an infinite loop looking for a queue entry. ANABOLISM will call INITIATE for independent runners. Thus, it performs the DOCTOR function for independent runners.

1. Wait for the event ANABOLEVENT to happen.

2. While there are queue entries the following is done.

   a. A PIB is created (CREATEPIB is called).

b. Put name and entry point SIRW into the PIB.

c. If there are no parameters the area is released. Otherwise, the area is saved to hold the parameters. The TASKPARAMS word of the PIB points to the area.

d. The priority, stacksize, box number and other information is placed in the PIB.

e. INITIATE is called passing the PIB.

3. Go to step 1.

OUTLINE OF ETERNALIR
------- -- ---------

ETERNALIR will perform periodic and queue driven functions for the MCP. There is a copy of ETERNALIR in each local box and in global.

1. Finish MCP initialization.

2. ETERNALIR will do the following periodic functions.

a. Select scheduled tasks to be initiated (call procedure SELECTION). If a task is to be initiated, INITIATE will be called or a message will be sent to the ETERNALIR that can initiate (see) the task.

b. Check tasks that have specified a WAITLIMIT or ELAPSEDLIMIT. If the limit is exceeded the task is DSED.

c. Check for peripheral status change.

d. Poll MCM fail registers.

3. Wait a specified time (based on where the ETERNALIR is running and the system type) or for the event for this ETERNALIR to be caused. If the event was caused, the queue driven functions (partial list) outlined below are done. The entire queue is processed.

a. Force a scheduled task to run (FS a task).

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

c. Terminate a task (call TERMINATE).

d. Log processor error.

e. Build or terminate the intrinsics stack in this box.

f. Shrink (during a freeze) or expand (during a thaw) a library stack.

g. Get stack history for PROCESSKILL when it does not have enough stack space to run.

4. Go to step 2.

SECTION 6

HARDWAREINTERRUPT PROCEDURE

INTRODUCTION

The procedure HARDWAREINTERRUPT77 provides the interface between the B7000 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 and other interrupts.

The section of the B7000 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 of the HARDWAREINTERRUPT77 procedure.

OUTLINE OF HARDWAREINTERRUPT77

1. If the processor number is equal to the stack number, the processor is in control mode 2 on an alternate stack (see step 7 for control mode 2 actions). Otherwise the processor is on a user stack. A case statement is executed based on interrupt class.

2. EXTERNAL interrupts. After the interrupt is processed an EXIT is performed.

   a. EGG TIMER. A memory dump is taken.

   b. Completed I/O's and status change. Call IOFINISH67.

   c. Processor to processor interrupts. Call LISTEN.

3. SPECIAL CONTROL interrupts. After the interrupt is processed an EXIT is performed.

   a. Stack overflow. If the stack is processing a stack overflow, a fatal memory dump is taken. Otherwise, the limit of stack is moved into the overflow area and KANGAROO is called to stretch the stack.

b. Interval timer. Procedure GEORGE is entered.

4. CLASS I and CLASS II interrupts. Not all interrupts are listed below. In general, only the interrupts that can be recovered (processed) are listed. After the interrupt is processed an EXIT is performed. If the interrupt can not be recovered, a branch is made to step 6.

   a. Segmented array. Call SEGARRAYINTERRUPT.

   b. Presence bit. Call procedure PRESENCEBIT.

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

   d. Invalid Operator. Determine if the interrupt is a true error or the user wants an MCP function performed. These invalid operators are set up by the compiler. The proper procedure will be called to perform the function requested. The types of functions performed are listed below.

      Linking to a library.

      Making an intrinsic reference for the first time.

      Opening a queue.

      Opening a file.

5. ALARM interrupts. Some of these interrupts are processed and an EXIT is performed. In other cases the stack is DSED.

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

   b. Memory parity or Memory fail 1. Call MEMORYERROR.

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

   d. Processor internal. Log the error and retry the instruction. If it fails the retry the stack is DSED.

6. Search for on-fault statement. If there is an on-fault statement it is executed. If the program does not have an on-fault statement the stack is DSED.

7. If the processor is in a control mode 2 stack the following is done.

   a. Log the error.

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.

c. If this is not the only CPM is control mode 2 a dead stop is performed. If this is the only CPM in control mode 2 a memory dump is taken.

SECTION 7

MEMORY MANAGEMENT

INTRODUCTION

Before a program can execute, it must have some memory areas assigned to it. These areas include major structures like the Stack, PIB and Segment Dictionary. In addition to these structures, the program must also have some memory areas for code and data. As a program runs it will dynamically allocate and deallocate additional memory areas.

Because all programs share a common memory resource, it must be managed by the MCP. Memory Management consists of processing user requests for memory, returning memory to the list of available memory, accounting for memory usage and providing memory protection. Memory Management also provides a virtual memory scheme. Using this scheme, programs may allocate more memory than is physically present on the system.

These subjects and many others will be discussed in this section. This discussion is followed by several procedure outlines. Additional information on Memory Management (Memory Management and Swapper) can be found in the System Software Support Manual.

VIRTUAL MEMORY CONCEPT

Memory Management is based on a scheme of virtual memory which allows programs to allocate memory beyond the physical memory limit. The virtual memory management scheme is based on two important aspects. The first is the difference between physical memory and virtual memory and the second is program segmentation.

PHYSICAL AND VIRTUAL MEMORY

For a given system, physical memory is a finite ammount of memory which is expressed in a number of words (one word is six bytes). This memory size is the maximum amount of memory that can be in use at any point in time. The MCP allocates memory for users from the pool of available physical memory. When a user is given a memory area, it is a part of physical memory.

An area of physical memory which has been assigned to a user will not always remain in physical memory. If the demand for memory exceeds the limit of memory, some areas of memory will be written out to disk. The process of writing the contents of a memory area out to disk is called an overlay. When the overlay is performed the users physical memory becomes an area of virtual memory. While an area is on disk the user does not have access to the data in the area. If the user requires access to the data a new area in physical memory must be found and the data read from disk.

At any point in time part of a program may be in physical memory while other parts of a program may be overlayed on disk. The parts of a program that are overlayed on disk are in virtual memory.

PROGRAM SEGMENTATION OF DATA AND CODE

The process of program compilation converts the symbolic form of a program into object code. The object code is placed into a code file. The code for a program is maintained as multiple code segments. This process of code segmentation is performed by the compilers. For each code segment the compiler generates a segment descriptor which "points" to the segment. The segment descriptors are maintained in a dictionary and kept in the code file. Thus, a code file contains both code segments and a segment dictionary.

When a program is executed, the segment dictionary is read into memory and placed in a Segment Dictionary Stack. This stack is never executed. It is only used to hold segment descriptors.

Each segment descriptor contains a length field, address field and a presence bit. The length field specifies the length of the code segment. The address field contains the address where the code segment can be found. The presence bit indicates where the code segment is located. If the presence bit is on, the code segment is in memory. In this case the address field contains a physical memory address (the base address of the code segment). If the presence bit is off, the code segment is on disk. In this case the address field

contains a record number into the code file (the record number of the code segment). The address field does not contain a disk address because the field is not large enough to hold a disk sector address and the compiler does not know where on disk the code file is located.

Data items from the program are also maintained as separate areas in memory. For example, an array in Algol or an item at the 01 level in COBOL will be managed as a data segment. Each data segment is "pointed" to by a Data Descriptor. Data descriptors are maintained in the programs stack (process stack). These data descriptors are built as the program executes stack building code. Stack building code is generated by the compiler as it scans the declarations or DATA DIVISION of a program.

Each data descriptor contains a length or index field, address field, presence bit, copy bit, indexed bit and size field. The size field indicates the size of the data items in the data segment. For example, the data items could be 4 bit characters 8 bit characters, words or double precision.

The index bit indicates the status of the length or index field. If the index bit is on, the length or index field is an index. This is called an indexed data descriptor. The index value is the offset above the base of the area. An index value contains both a word offset and a character offset. If the index bit is off, the length or index field is a length.

If the length or index field is a length (the index bit is off), it specifies the length of the data segment. The length field is based on the item size. For example, if the size field is 8 bit characters and the length field is 50, the data descriptor represents an area which is 50 characters long. If the size field is words and the length field is 50, the data descriptor represents an area which is 50 words long.

The address field contains the address of the data segment. The presence bit indicates where the data segment is located. If the presence bit is on, the data segment is in memory. In this case the address field contains a physical memory address (the base address of the data segment). If the presence bit is off, the data segment is on disk. In this case the address field contains a record number into the overlay file (the record number of the data segment). The address field does not contain a disk address because the field is not large enough to hold a disk sector address.

If the copy bit is off, it specifies that the data descriptor is a MOM descriptor. For a given data area there will be one MOM descriptor. All other descriptors that point to a data area are copy descriptors (the copy bit is on).

When a data area is on disk, the address field of the MOM

descriptor will point to the data in the overlay file. The address field of any copy descriptors will point to the MOM. That is, the address field of any copy descriptors will contain the physical memory address of the MOM descriptor. When the data area is read into memory, the MOM descriptors address field will be updated to point to the physical memory address where the data resides. The copy descriptors may or may not be updated to point to the memory area. This depends on several factors which are outlined under the discussion of PRESENCEBIT.

MEMORY STRUCTURE

The structure of memory determines how the MCP looks at and sees memory. The structure also determines what physical memory a program can use as well as how programs can communicate with each other. Memory structure has evolved with the Large Systems family of computers and to a large degree is dependent on machine type. However, some Large Systems have the ability to structure memory in two different ways. The three ways memory can be structured are Monolithic, Tightly Coupled and Extended Memory.

MONOLITHIC SYSTEM

The monolithic memory structure was the first memory structure used on the Large Systems family. It is a simple, but limited memory structure. Systems which use this memory structure are the B5900, B6800, B6900 and B7800.

The software sees a monolithic system as having a single block of memory. This block can be at most one million words (6 MB). The limit of one million words is imposed because the address field of a data descriptor or a segment descriptor is 20 bits. All processors (multi-processor B7800) and I/O processors (multi-IOM B7800) have visibility to this memory. That is, memory is global to the processors. The B7800 is the only system which allows multiple processors under the monolithic memory structure. Figure 7-1 is how the MCP sees memory under the monolithic memory structure.

Figure 7-1. Monolithic Memory Structure

All programs and the MCP share this block of memory. They all have full data visibility which means programs can pass data to other programs without restriction. Each word in memory has a unique memory address. Programs also have full visibility to any processor or I/O processor. Thus, programs can be processed by any available processor (B7800 multi-processor system) and they can issue I/O operations to any I/O processor (B7800 multi-IOM system).

The advantages of the monolithic memory structure are that it is a simple structure and all programs have full visibility to other programs and all processors.

The disadvantages of this memory structure are only a B7800 can have multiple processors (CPM's and IOM's) and memory is limited to 6 MB. The single processor and I/O processor restriction is not a function of monolithic memory. In order to have multiple processors that can all see the same memory, it must have multiple access ports. The B7800 is the only system that is constructed with multiple port memory.

TIGHTLY COUPLED SYSTEM

Tightly Coupled systems are an extension to the monolithic memory structure. A Tightly Coupled system allows more than one million words and multiple processors on B5900, B6800 and B6900 systems. Although a Tightly Coupled System allows more memory, it imposes visibility restrictions on the processor and I/O processor. Systems that are capable of being Tightly Coupled are the B5900, B6800, B6900 and B7800.

A Tightly Coupled system has a portion of memory called GLOBAL Memory ("GLOBAL Memory" is a trademark of Burroughs Corporation) that is visible to all processors and I/O processors. Another part of memory is called Local Memory and is only visible to one processor and I/O processor. The limit of processor adressability is still one million words on a Tightly Coupled system but part of it is in Global and part of it is in Local.

The combination of some Local memory, a processor and an I/O processor is called a Box. As stated above the Local memory is visible to only one processor and I/O processor. Because the B7800 has multiple port memory, a Local Box may have more than one processor and I/O processor. A B7800 system is only member of the Large Systems family which allows more than one processor or I/O processor in a Box.

A processor in one Local Box is restricted from looking at the memory in another Local Box. This is because physical memory addresses are duplicated in the Local Box. Another restriction is that a program running in Global memory can not look into (try to fetch a word from) a Local Box. The processor would not know which Local memory to use.

All Tightly Coupled systems except the B7800 require the use of a Global memory cabinet. This cabinet contains logic for locking specified processors out of memory. It also contains some amount of physical memory. The B7800 systems simply use another MCM as the Global memory cabinet. A B7800 system can use a MCM for the Global memory cabinet because it already has memory lockout and address range switches.

The Global and Local memory areas are distinct. A memory area is not allowed to extend from a Local Box into any other Local Box or Global. Each of these memory Boxes is managed independently. Therefore, a system could be out of memory in one Box but have memory in other Boxes. The fact that there is memory in other boxes would not help the Box that was out of memory. Figure 7-2 is how the MCP sees memory under the Tightly Coupled memory structure.

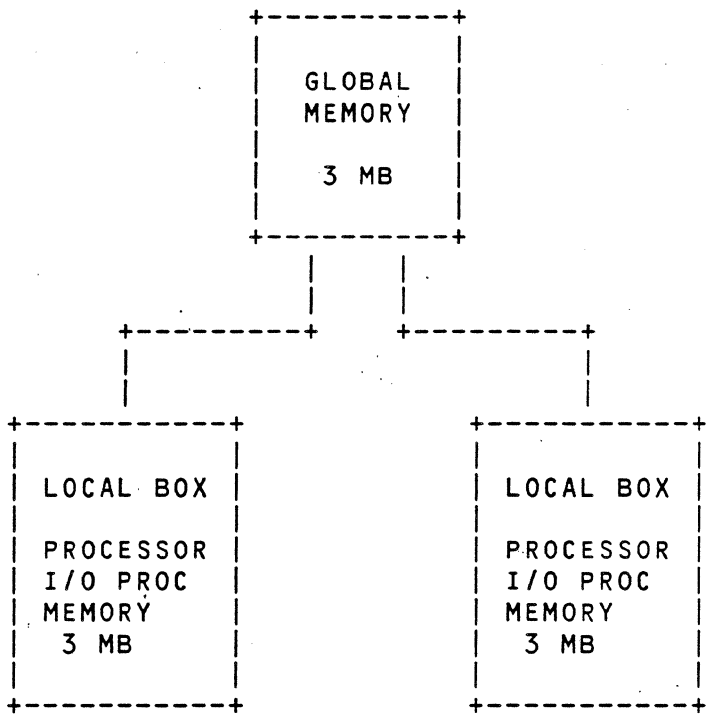


Figure 7-2. Tightly Coupled Memory Structure

When looking at Figure 7-2, it should be noted that all Boxes need not have 3 MB. That is, Global memory or a Local Box could be less than 3 MB. The figure was shown with 3 MB in each because most users configure memory this way. There is a restriction that any Local Box and the Global Box can not exceed 6 MB. This restriction is imposed because a data descriptor and a segment descriptor has a 20 bit address field which will only address one million words (6 MB).

On a Tightly Coupled system it is very important to avoid congestion of Global memory and to keep tasks balanced across the Boxes. More information on this subject is in the section titled MEMORY CONTROLS.

A Box is identified by its Box Number which is assigned by the MCP. Global memory is identified by the letter "G" or the word "Global". This letter or number is used extensively in the operator interface. For example, subsystems specified through the MS (make subsystem) ODT command are defined in terms of Box Numbers. In the CU display, "LOCAL MEMORY 1" refers to the Local memory of Box number 1. In the list of active entries generated by the A (Active entries) ODT command, the number on the extreme left of a mix entry is the Box number being used by that task.

The primary advantage of Global Memory is that of additional memory. Because Global memory has multiple access ports it

also allows more than one processor to be placed on a B5900, B6800 and B6900.

The primary disadvantages are the contention that can take place in Global memory. The workload can also become unbalanced among the Boxes from a memory and processor standpoint. Remember that memory in each Box in managed independently and a processor can only run programs in the Box that the processor is in. A B7800 Tightly Coupled system has a further restriction that limits I/O visibility. Peripheral units can only be assigned to one Box (they may not be exchanged across Boxes). Therefore, if a program running in one Box wants issue an I/O to a peripheral in a different Box, the I/O must be done out of Global memory.

EXTENDED MEMORY SYSTEM

The concepts of the Extended Memory System are a logical extension of the Tightly Coupled System. The Extended Memory System follows some of the same rules as a Tightly Coupled System, but it has fewer restrictions. Currently the only Extended Memory Systems are the A3, A9 and B7900. There are differences in the Extended Memory implementation on these three systems which will be covered as they come up.

A basic concept of the Extended Memory architecture is that of the Address Space. On an Extended Memory System, memory is seen through the software as some number of address spaces. The maximum number of address spaces allowed on the B7900 is fourty-six and on the A9 and A3 is sixteen.

Each of these address spaces can contain up to one million words (6 MB). The B7900 allows an address space to be less than one million words. In addition, the B7900 allows address spaces to be different sizes. On the A9 and A3 the size of the address space is fixed at one million words with the possible exception of the last address space.

An address space is the memory that a task may access at any one time. Because the MCP needs to control the entire system and because tasks in different address spaces may need to share code and data, some portion of memory must be common to all address spaces. This memory must be always visible (available) to all of the tasks no matter where they are running. This portion is known as the shared component of the address space. Because the non-shared portion of an address space is unique to that address space, this portion is known as the local component of the address space. A system which contains 2 million words (12 MB) could contain four memory components, each containing one-half million words. One of these components would become the shared component and the other three would become local components. See figure 7-3.

The combination of the shared component plus one of the local components makes up one address space. Therefore an address space is an addressing continuum of up to one million words. The shared component maps onto the lower portion of each continuum and has environment relative addresses ranging from 0 to N. The local components each map onto the upper portion of (separate) environments and have environment relative addresses ranging from M to one million, where M>N. See figure 7-3. The two environments are distinct and a memory area can not span from a local component into the shared component. The unit of granularity for the size of a memory component is a "page" which is 128K words.

There is a unique addressing space that consists of only the shared memory environment component. Or, equivalently, it can be thought of as the unique addressing space that has a null local component. All other addressing spaces contain both the shared component and a local component.

An address space is identified by its Address Space Number (ASN) which is assigned by the MCP. This number is used extensively in the operator interface. For example, subsystems specified through the MS (make subsystem) ODT command are defined in terms of ASNs. In the CU display, "LOCAL MEMORY 1" refers to the local component of the address space with an ASN of 1. In the list of active entries generated by the A (Active entries) ODT command, the number on the extreme left of a mix entry is the ASN of the address space being used by that task.

The shared component is often referred to as "Global" or "G" in documentation and system interfaces to be consistant with Tightly Coupled systems. The ASN Extended Memory software is an evolutionary development from the Tightly-Coupled software and as such, uses many of its interfaces and conventions, often in enhanced and less restricted forms.

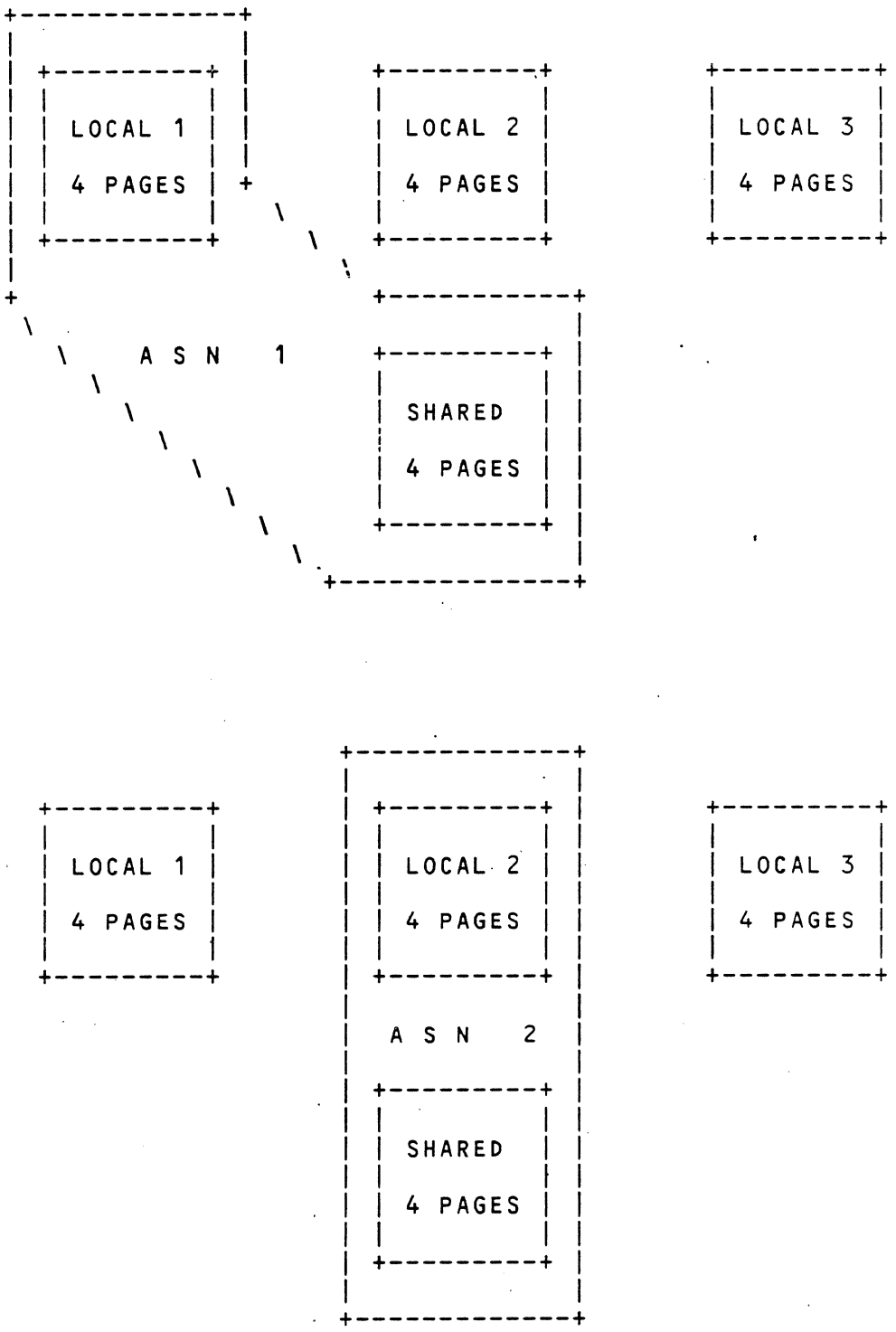


Figure 7-3. Extended Memory Structure (1 of 2)

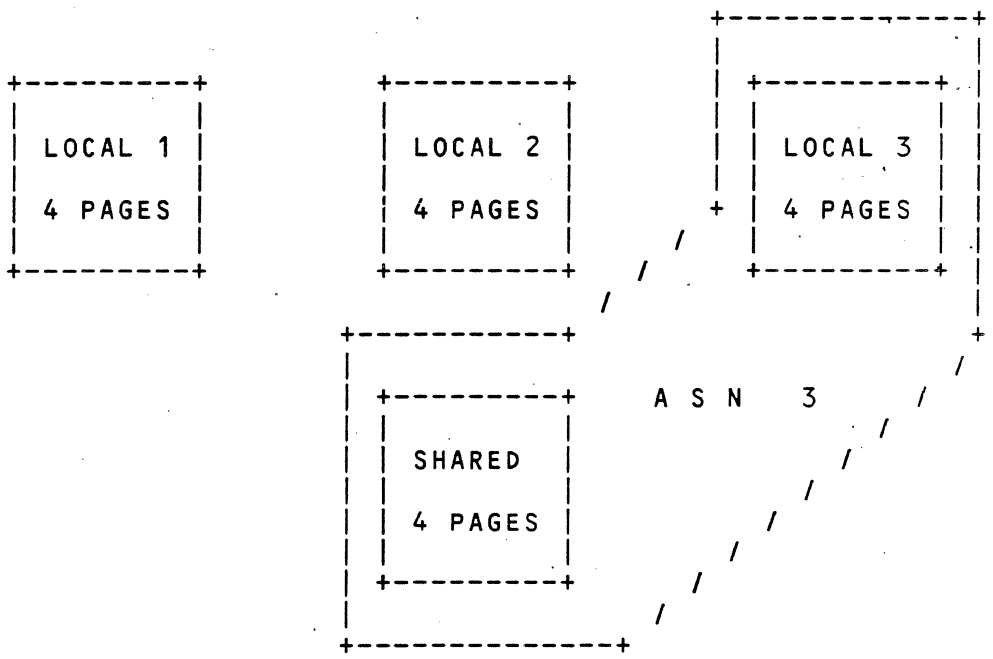


Figure 7-3. Extended Memory Structure (2 of 2)

The ASN of a task determines which environment number the processor will use when executing a task. The combination of environment number and 20-bit environment relative address is decoded by the memory subsystem (hardware) to a page number and a page relative address. The environment number is also used by the I/O processor for IO's of the task.

An Extended Memory system is not a Tighlty-Coupled system, as its requestors (processor and I/O processor) can each access all of memory. The requestors have a monolithic organization: any task can run on any processor (B7900 multi-processor system) and any I/O processor (B7900 multi-HDU system) can perform input/output operations for any task. On a B7900 this monolithic requestor organization allows the same failsoft redundancy of previous monolithic B7000 systems.

To take maximum advantage of the extended memory systems large memory space, tasks must be allocated among the address spaces to achieve a balanced processing load with minimal memory contention. Although the MCP does much of this automatically, depending on the mix, some specific tailoring may be necessary to obtain optimum performance. More information on this subject can be found in the MEMORY CONTROLS section.

The primary advantages of this memory architecture are large amount of memory and the fact that programs are visible to any processor or I/O processor. In addition, the Extended Memory architecture allows greater reconfiguration of memory.

The disadvantages are program (process family) visibility and

contention that can occur in the shared component.

CODE AND DATA ENVIRONMENTS

A B7900 (not an A3 or A9) processor distinguishes code memory references from those of data by using a different environment number for code than that of data. The combination of environment number and 20-bit environment relative address is decoded by the MSMs to a page number and a page relative address. The Addressing Space Number of the task determines which environment numbers (code and data) the CPM will use when executing that task as well as which environment numbers the HDUs will use for IOs of the task.

ASNs can be configured in four different ways. These configurations are described in the following paragraphs.

The Environment Component (EC) for code and the EC for data (in each portion of the ASN) can be mapped onto the same physical memory. This results in no distinction in the addressing of code from that of data. The maximum size of an ASN under these conditions is 6 MB (1 million words) and is shown in figure 7-4.

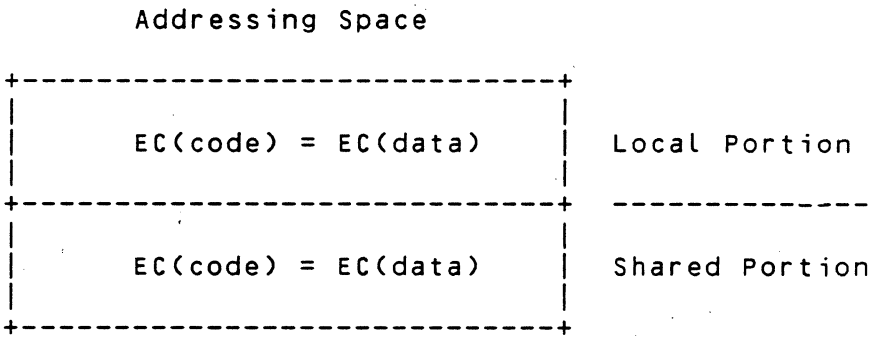


Figure 7-4. Code and Data Areas Combined

The code environment can be separated from the data environment. If this is done the size of an ASN can be as large as 12 MB (2 million words) of physical memory. This implementation is accomplished by providing that an ASN map onto 2 separate 6 MB environments: one for code and one for data.

The code and data split can be performed in three ways. These three methods are shown in figures 7-5 through 7-7. In these figures each small box is an Environment Component (EC), a distinctly managed subdivision of the total addressing space which is then further divided by the hardware into some number of 128k word pages.

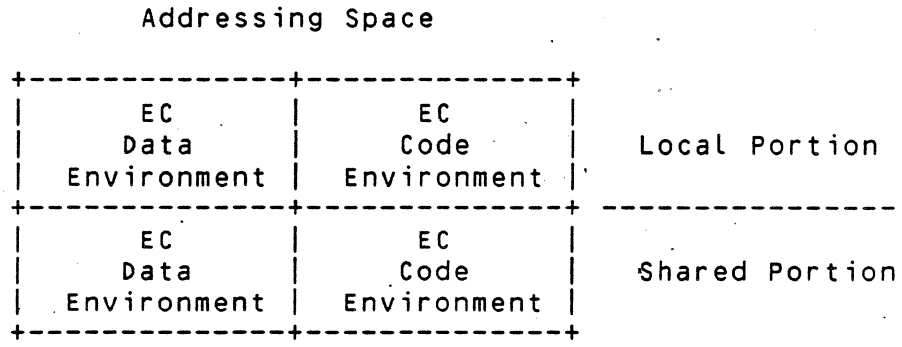


Figure 7-5. Split Shared and Split Local Environment

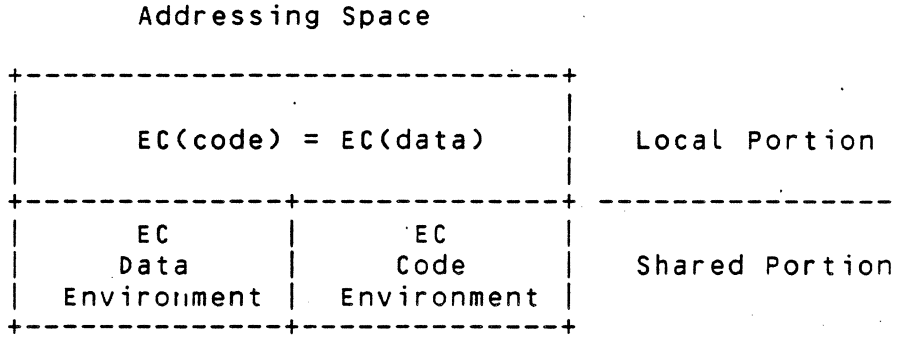


Figure 7-6. Split Shared and Combined Local Environment

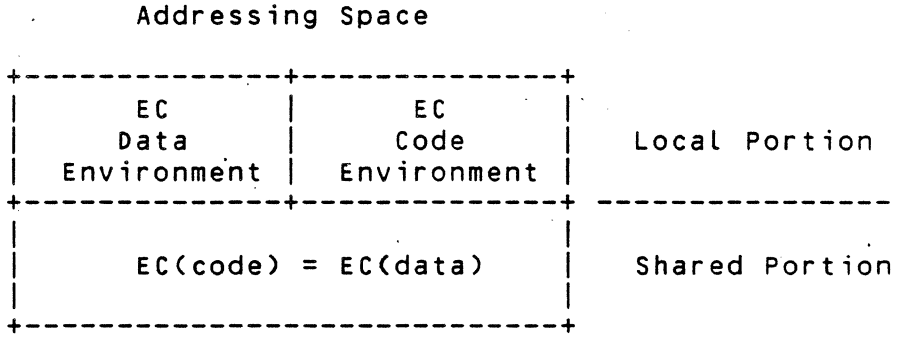


Figure 7-7. Split Local and Combined Shared Environment

Separate code environment software is fully compatible with all user software including DMSII. However, SWAPPER cannot be run within an ASN that has a separate local code component and Working Set functions apply only to the data portion of such an ASN. When performing dump analysis or reading program dumps pertaining to such an ASN it should be remembered that code and data areas may have the same 20-bit address yet still represent physically distinct memory.

No overlap of a local with shared is permitted, even when the environments are distinct. Thus, no shared address can be the same as any local address, be it code, or data. For example, if Shared were configured as 4 PAGES DATA 3 PAGES CODE, then 4 PAGES DATA 4 PAGES CODE are the maximum that could be specified for any local; 4 PAGES DATA 5 PAGES CODE is not allowed.

The EC ODT command and configuration file can be used to specify the assignment of pages to code and data environment components.

Respecifying an existing ASN local that has no separate code environment into one that does (splitting the local environments) or going the other way, rejoining a split local into one with equivalent code and data ECs, is accomplished only by deactivation (with the EC ODT command) and reactivation of the Local part of an ASN. Because Shared cannot be deactivated, such respecification can only be accomplished by way of a HALT/LOAD (the DEFER option on the EC ODT command is used).

Expanding the code environment of a component requires the reallocation of the code space management structure that is a save area within the data EC. It may not be possible for the system to achieve this if the ASN concerned is heavily loaded. Note that the code management structure is quite small in comparison to the amount of usable memory gained by employing the feature. The code environment can be reduced without special considerations.

MEMORY MANAGEMENT STRUCTURES

Reguardless of the structure of memory certian software structures are required to manage memory. The structures used in the MCP are called memory links. These links are used to form a pool of available memory and protect in use memory.

The memory links divide memory into three basic types.

MEMORY TYPES

A given memory area can be available. If it is available, it is not in use but is in a memory list or pool. If a user makes a request for an area it will be allocated out of this pool of available memory.

A given memory area can be overlayable. If it is overlayable, it is in use but can be overlayed (written to disk). The process of overlaying is done when there is not enough physical memory for user demands.

A given memory area can be save. If it is save, it is in use and may not be overlayed or moved. It is important to note that save means two things. First it means the area will never be written to disk. Second it means the area will never be moved around in memory. In general, memory areas are moved when memory becomes checkerboarded.

MEMORY LINKS AND LISTS

Available memory areas are linked together with memory links which are words constructed so that the linked list lookup (LLLU) operator may be used to find an available area. Memory links precede and follow all memory areas.

An available area has four memory links which are named:

AVAILA

AVAILB

<THE AVAILABLE AREA IS BETWEEN AVAILB AND AVAILY>

AVAILY

AVAILZ

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 once a save area is allocated it can not be moved to another location in memory. The implementation will also avoid two save areas being close together with a small area between them. This is undesirable because the save areas may not get removed (deallocated) for a long time and the small area may be too small to be usable. Thus, it is effectively wasted memory.

Overlayable memory areas are also kept together. The main reason they are kept together is to keep them away from the save areas. Overlayable areas are not kept as tightly packed

as save areas because they can be moved around in memory if needed.

The way this is implemented is to link the available areas into two different lists. One list is for areas that will be most efficiently used for save memory, and the other list is for 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. An available area is linked into the (to be used as) overlayable list, if it adjoins an overlayable area. Notice that with these two rules all available areas will be linked into at least one list, and that an available area might be linked into both lists. See figure 7-8.

The start of the memory lists are pointed to by a word named MEMBASEPLACE in the BOXINFO array. To make this discussion more readable assume that a varable called MEMBASE is set as follows: MEMBASE:=BOXINFO[BOXNUMBER,MEMBASEPLACE].[19:20]. Where BOXNUMBER is the Box (or memory component) the program is running in. The word at MEMORY[MEMBASE] is the head word for the overlayable memory list. The word at MEMORY[MEMBASE+1] is the head word for the save memory list. The word at MEMORY[MEMBASE-1] is used as a lock (MEMLOCK) for the links. Figure 7-9 is a layout of the memory link head words.

The AVAILA word is used as a forward link and the AVAILB word is used as backward link for the overlayable list. The AVAILZ word is used as a forward link and the AVAILY word is used as backward pointer for the save list. Figure 7-10 is a layout of the available link words and figures 7-11 and 7-12 show how the areas are linked. Notice that although the A-B and Y-Z links are associated with the same area, they never actually point to each other.

The linked list of available memory areas that are to be used as save memory (save list) is linked in size order (smallest to largest). See figure 7-13. Thus, when obtaining a save memory area a best fit algorithm is used (to avoid memory checkerboarding). This also implies a best fit algorithm is used when returning a memory area to the list. So a linked list lookup must be done to place it in the linked list.

The linked list of available memory areas that are to be used as overlayable memory (overlayable list) is linked in age order. The most recently released area is first. See figure 7-13. Thus, when obtaining an overlayable memory area a first fit algorithm is used (for speed). When an area is returned to the list a linked list lookup is not required. The area is simply placed at the beginning of the linked list.

There is one set of links for each Box (or memory component) configured. Thus, on a monolithic system there are two lists

(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 lists in each users swap area. The set of links for a given program is pointed to by the MEMLOC word in the base of the stack. In all cases the links look and function the same.

There are four links around in use areas which are given the following names:

LINKA

LINKB

LINKC

<THE IN-USE AREA IS BETWEEN LINKC AND LINKZ>

LINKZ

These links are not actually linked to each other. Instead the links are used to contain information about the in use area. The layout of the in use links is shown in figure 7-15.

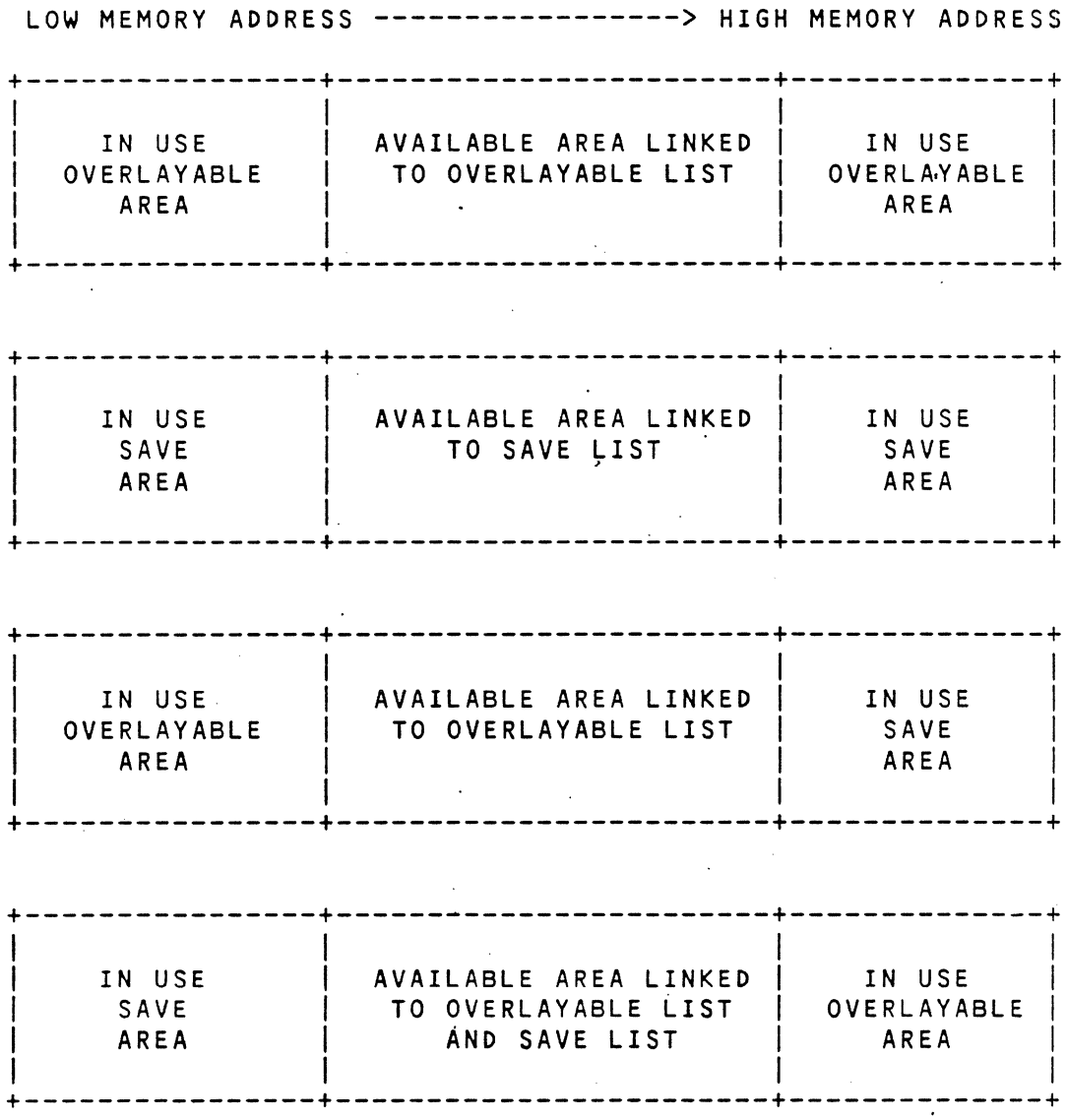
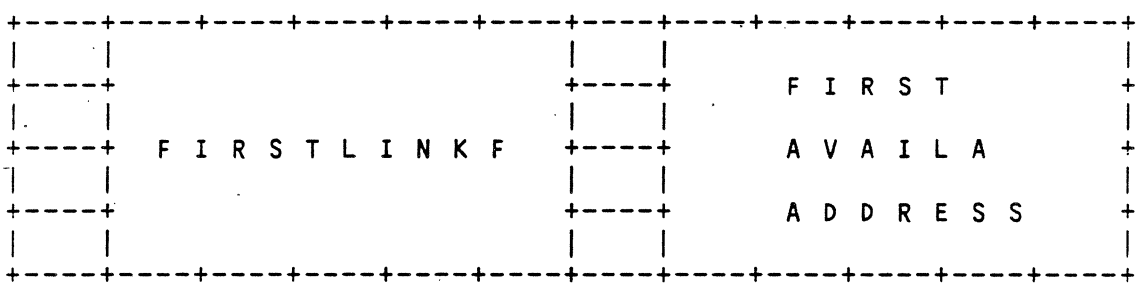


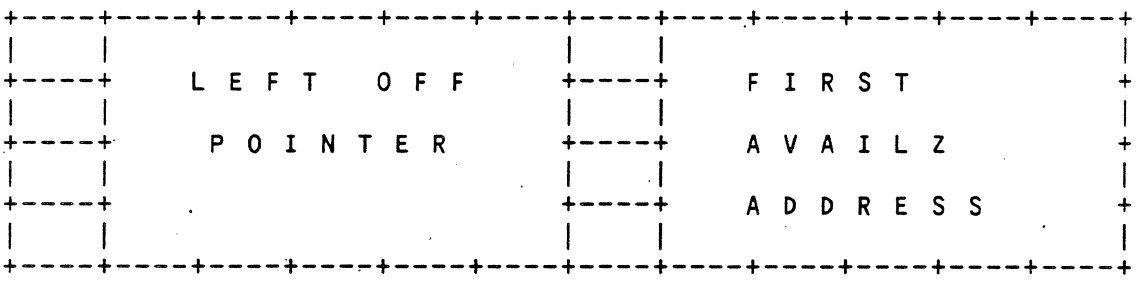
Figure 7-8. Memory Links

OVERLAYABLE LIST HEAD WORD



HEAD WORD IS LOCATED AT MEMORY[MEMBASE]

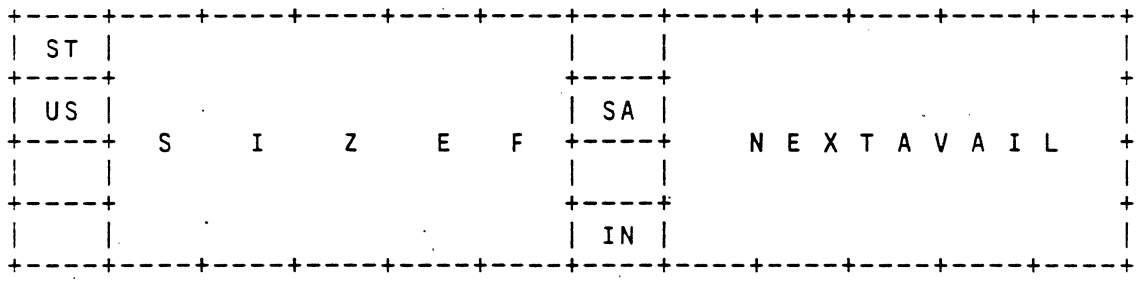
SAVE LIST HEAD WORD



HEAD WORD IS LOCATED AT MEMORY[MEMBASE+1]

Figure 7-9. Memory Link List Head Words

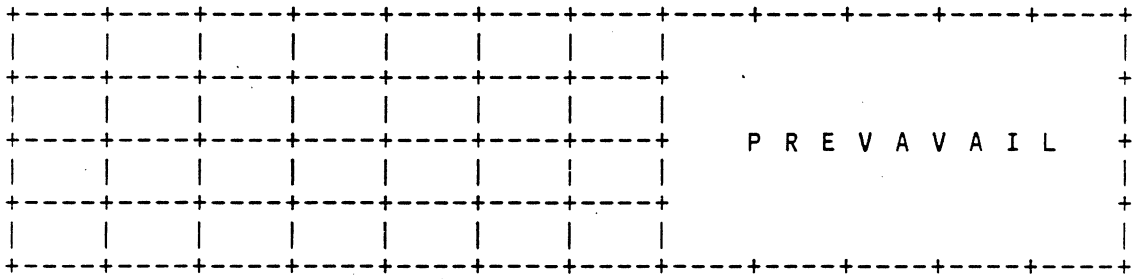
AVAILA TAG 1



[47:01] STOPPERF
[46:01] USABLEF
[43:20] SIZEF
[22:01] SAVEDF
[20:01] 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.

AVAILB TAG 0

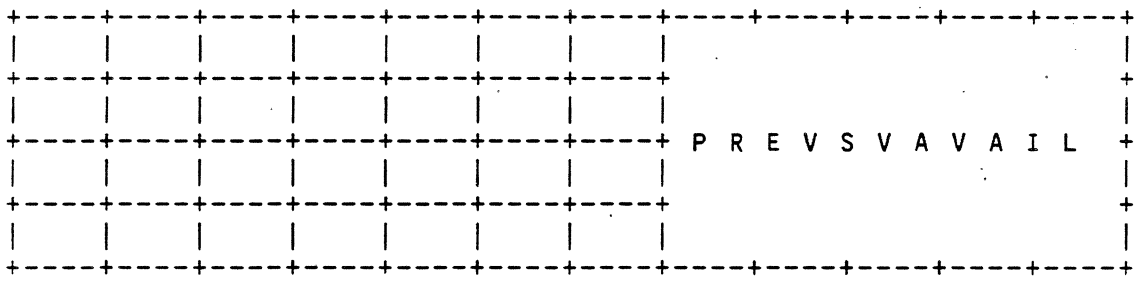


[19:20] PREVAVAIL

PREVAVAIL — LINK TO PREVIOUS AVAILABLE AREA IN OVERLAYABLE CHAIN. THAT IS, A BACK LINK.

Figure 7-10. Available Memory links (1 of 2)

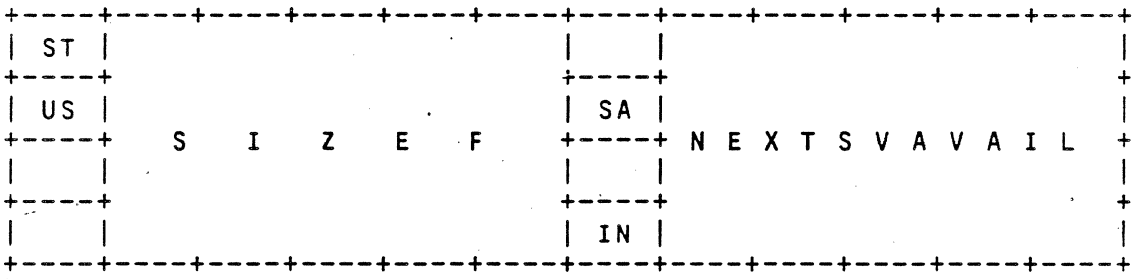
A V A I L Y T A G 0



[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



[47:01] STOPPERF
[46:01] USABLEF
[43:20] SIZEF
[22:01] SAVEDF
[20:01] 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.
NEXTSVAVAIL LINK TO NEXT AVAILABLE AREA IN SAVE CHAIN.

Figure 7-10. Available Memory links (2 of 2)

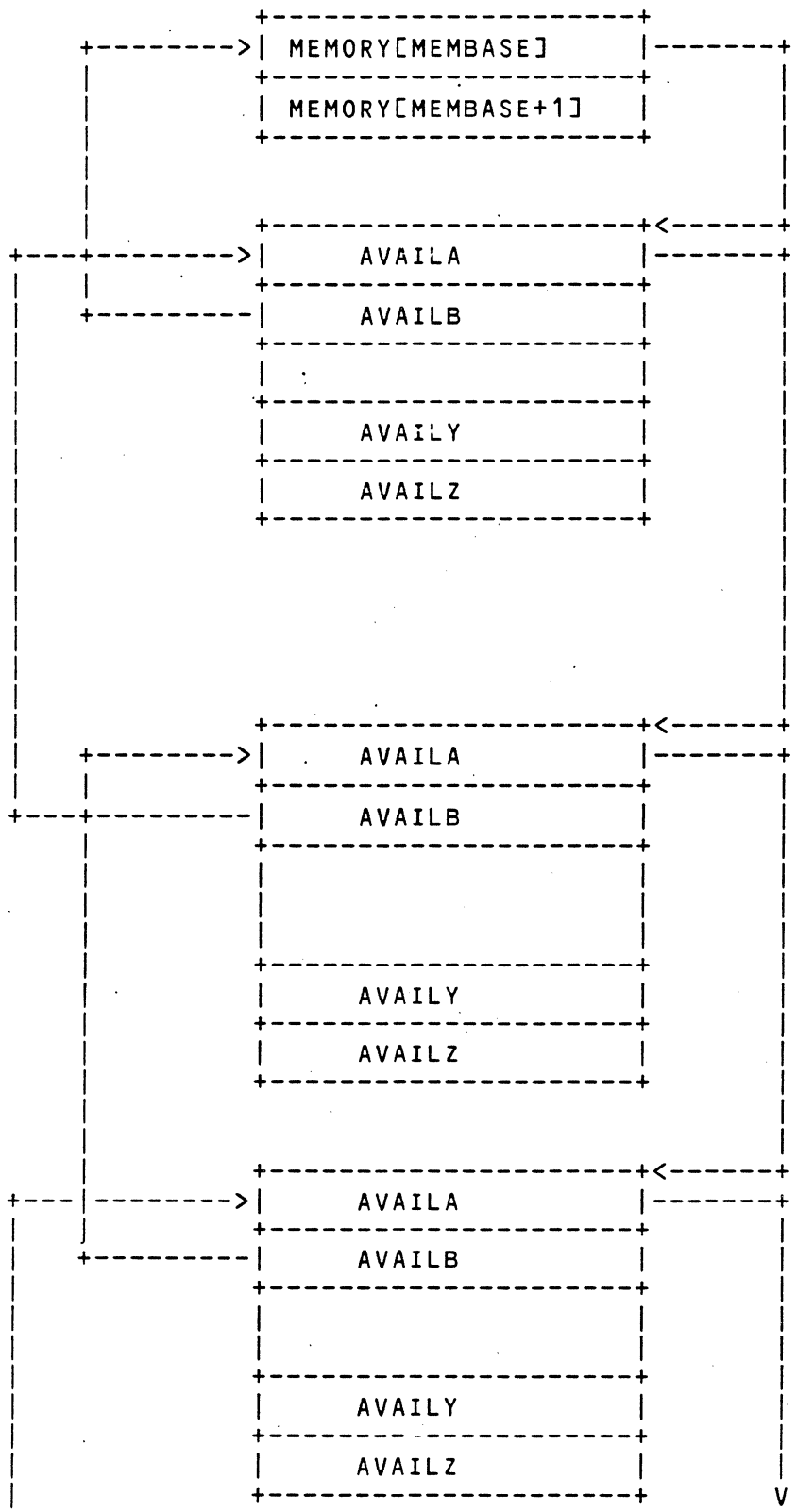


Figure 7-11. AVAILA and AVAILB Links

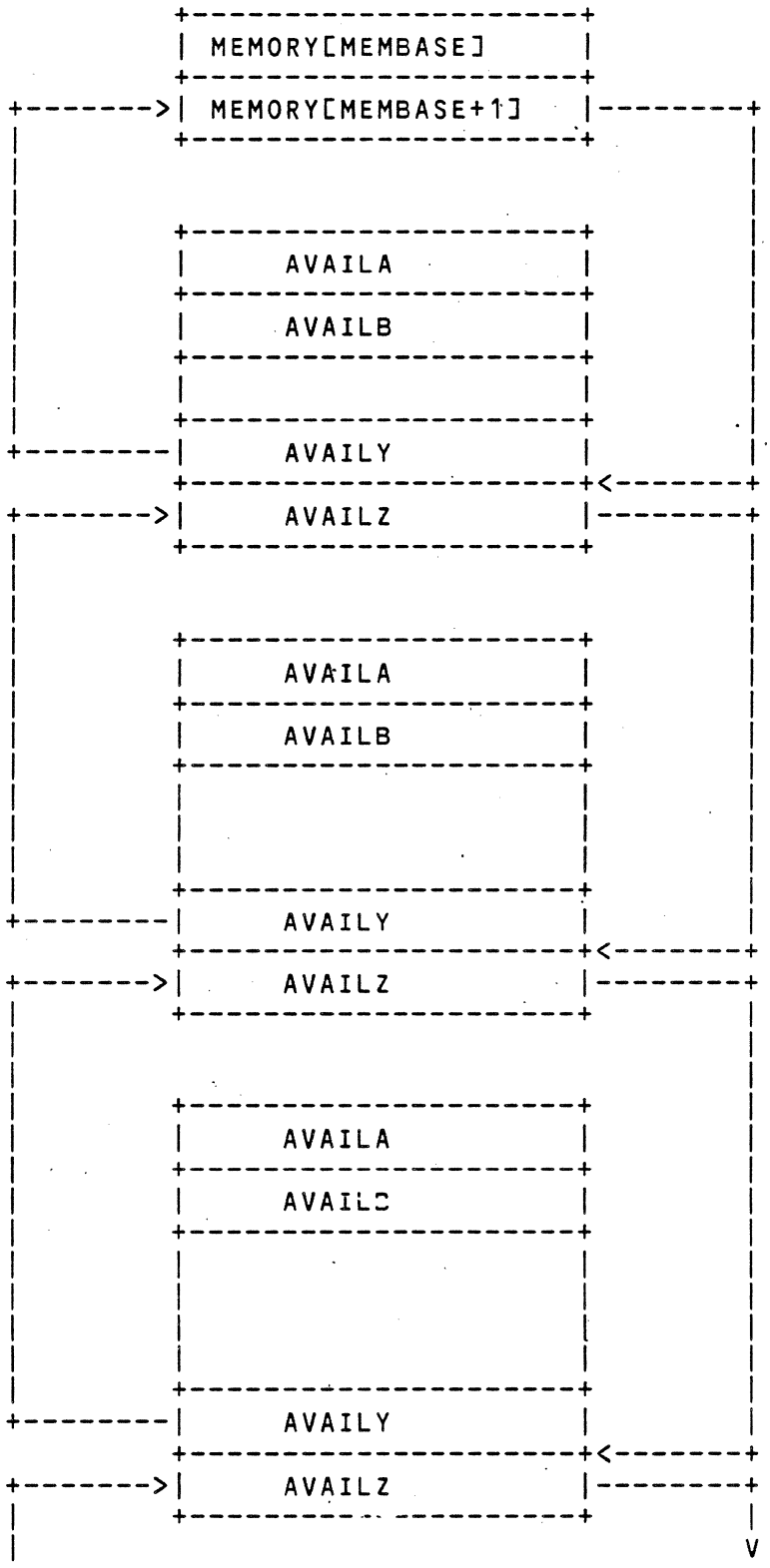


Figure 7-12. AVAILY and AVAILZ Links

subsystem outside the named set. One firm rule is that a fully dependent task may not be in shared memory if its parent is in the local component of an ASN.

Whenever a SUBSYSTEM has been specified but had to be disregarded, a warning message is displayed (and logged) at the start of the task.

DASDL CONTROLS

The database administrator can specify that the database stack for a particular database is to run in a given memory subsystem. This specification is made by having a value for the SUBSYSTEM task attribute compiled into the ACCESSROUTINES code file.

When the database stack is initiated, it will be placed in the shared memory component if the subsystem contains more than one ASN and VISIBILITY was not specified as MINIMAL. Otherwise, it will be put into one of the local ASNs in the subsystem.

If it is desired to have the database stack in a local ASN, that ASN and no other should be included in the subsystem definition. If this is not done operational difficulties will result. If the database stack is in one local ASN and database access programs are initiated in other local ASNs, they will be DSed due to lack of visibility to the database stack. During the first or any subsequent OPEN of the database, if the ASN in which the database stack is placed is not visible to the program trying to OPEN the database, the program will get a DMOPENERROR exception on the OPEN statement. There is at most one database stack for a given database present anywhere on the total system at any point in time. It is possible to determine which ASN the database stack is running in by using the ODT command "DBS".

CANDE CONTROLS

Semidependent tasks permit a parent running in one ASN of a system to have offspring in other ASNs, subject to certain visibility constraints. The user tasks spawned from CANDE meet these requirements, but certain actions must take place to allow the system to run these offspring to the fullest advantage. The term "offspring" refers to user tasks (i.e., COMPILE, EXECUTE, UTILITY).

There are two reasons for running CANDE in the local component of an ASN. First, it may be desirable to limit CANDE and all of its offspring to that component to permit other ASNs to run free of interference from that source. Second, CANDE may be

run in the local component of an ASN to reduce utilization of the shared memory component by CANDE.

In either of these cases, the site must take some overt action to allow or prevent the scattering of the offspring. Because of the current semantics of the SUBSYSTEM attribute as applied to CANDE, the default is to run offspring in the same subsystem as CANDE, and the CANDE run-time option SCATTER (#13) is set to allow CANDE's offspring to run outside of that subsystem.

If SCATTER is set, CANDE sets the tasker stack's SUBSYSTEM to null; this allows the MCP to pick any appropriate ASN at task initiation time. If the option is reset, CANDE sets the tasker stack's subsystem to its own (main CANDE stack) value.

The setting of this option is maintained in the tankfile in the same manner as the other options. If the value of SCATTER is changed, the effect of the change does not occur until the next initiation of the tasker stack.

Following are five possible ways to run CANDE to achieve various results. The method selected may be controlled by site management to achieve optimum performance in their environment.

1. Run CANDE in shared memory; scatter both the worker stacks and user tasks to all the local ASNs. This is the default.

2. Run CANDE in shared memory; scatter offspring in some local ASNs. Compile CANDE with an appropriate subsystem specification with default VISIBILITY.

3. Run CANDE and all the workers in any one of several local ASNs, but scatter the user tasks among all those ASNs. Compile CANDE with an appropriate subsystem specification and with VISIBILITY=MINIMAL.

4. Run CANDE and all offspring in a single local ASN. Compile CANDE with an appropriate single-ASN subsystem.

5. Run CANDE and the workers in a single local ASN, but scatter user tasks across the system. Use method 3 or 4 and set the SCATTER option to avoid propagating CANDE's SUBSYSTEM attribute.

Note that SYSTEM/CANDE can be compiled or bound once with a special SUBSYSTEM specification, e.g., "CANDESUBSYS" and with VISIBILITY set to MINIMAL. Then the use of CANDE can be tailored to be method 3, 4, or 5 above by changing the definition of CANDESUBSYS, using the "MS" ODT command and by setting or resetting the SCATTER option.

The capability to run CANDE in local with the workers in

several local ASNs cannot be offered, because the worker processes are fully dependent and use their visibility into the parent stack.

OUTLINE OF GETSPACE

This procedure is called to find memory space. It will return the address of the area if one is found, otherwise it will return a zero. The five parameters passed to GETSPACE are:

Size of area needed (not including links).

Stack number of MOM descriptor.

Options which specify save or non-save memory and other fields. These fields include:

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, 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 requested 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.

Absolute address of MOM descriptor.

Word which specifies which set of links to use.

1. The base of memory is set up from parameters. The base of memory tells GETSPACE which links to use. The base could point to a local box, global or a subspace.

2. Add 4 to the size passed to GETSPACE (for links).

3. Lock MEMLOCK if this is a non recursive call.

4. Compute the memory priority for the area. The priority is based on ODT priority, factor 4 and current time of day.

5. A Link List Lookup (LLLU) is done on the proper memory chain to find an area large enough. If the request is for

overlayable memory the chain at the base of memory is used. If the request is for save memory the chain at the base+1 of memory is used.

6. If the area found has the STOPPERF on, the end of the memory chain was found. The links are constructed such that a -1 is never returned by the link list lookup. The last link has a large size (STOPPERF on).

7. If an area was found, it is delinked from the chains as follows.

   a. If this is a recursive GETSPACE call and the area found is the same as the area the other copy of GETSPACE is working in, the area can not be used. GETSPACE branches back to do the link list lookup from this point forward.

   b. Do accounting to keep track of save and overlayable memory by box.

   c. The area is delinked from any memory chains it is in. The area can be in one or both chains.

   d. WSJUDGE is called to update the memory integral for the stack that owns the mom.

   e. If the area has more words than were required, the extra words are returned. These words must be linked into the proper chain or chains (see figure 7-4).

      If one word is returned, a one word link is built but not placed in the memory chains. If two words are returned, two links are built but not placed in the memory chains. If three words are returned, the two end links are built but not placed in the memory chains. If four or more words are returned, the links are built and placed in the memory chains.

   f. The memory links (LINKA, LINKB, LINKC and LINKZ) are built.

   g. If this is not a recursive call MEMLOCK is unlocked.

   h. If this is a recursive GETSPACE call the number of overlays in process is decremented. If there is an attempt to stop all overlays and the number of overlays in process is zero an event (NULLOLAY) is caused.

   i. GETSPACE returns the address of the area.

8. This is the end of the code where the area was found. The next section of code will try to find the area by doing overlays.

9. If GETSPACE was not to lose control of the processor a zero is returned.

10. If overlay activity is to stop this stack will wait on an event (WAKEOLAY) before it starts the overlay process.

11. The number of overlays in process is incremented.

12. If the request is for overlayable memory, branch to step 15. Otherwise the request is for save memory.

A loop is started to find some memory to overlay. The start of the loop is the FIRSTLINKF. The end of the loop is the upper end of memory as defined by the MEMLOC word passed to GETSPACE. The loop will look at sequential memory areas. The logic for one link is as follows:

```
IF THE AREA IS INUSE THEN
   IF THE AREA IS NOT IN A MOD THAT IS BEING SAVED THEN
      IF THE AREA IS SAVE THEN
         IF THE AREA IS STICKY AND NOT BEING MOVED THEN
            IF WE CAN GETSPACE FOR AREA THEN % NOTE 1
               SAVE THE ADDRESS OF THE NEW AND OLD AREA
               IT CAN BE TAKEN
            ELSE % NOTE 3
               MUST START OVER
         ELSE % NOTE 3
            MUST START OVER
      ELSE % IT IS OVERLAYABLE
         IF THE AREA HAS A SLOT IN THE OLAY FILE OR IT IS
          FROM THE CODE FILE THEN
            IT CAN BE TAKEN
         ELSE
            IF CAN FIND SPACE FOR AREA IN OLAY FILE THEN % NOTE 2
               PLACE THE ADDRESSES IN THE LINK WORD
               IT CAN BE TAKEN
            ELSE % NOTE 3
               MUST START OVER
   ELSE % NOTE 3
      MUST START OVER
ELSE % NOT IN USE
   IF THE AREA IS USABLE THEN
      IT CAN BE TAKEN
   ELSE % NOTE 3
      MUST START OVER
```

1. WE NEED AN AREA TO MOVE THE STICKY AREA TO. THIS IS A RECURSIVE CALL TO GETSPACE. GETSPACE IS TOLD NOT TO LOSE CONTROL OF PROCESSOR.

2. FINDOLAYSPACE IS TOLD NOT TO LOSE CONTROL OF PROCESSOR.

3. ALL REPLACEMENT STICKY MEMORY AREAS MUST BE RELEASED. LOOP STARTS OVER FROM THIS POINT.

13. This loop continues until enough areas have been picked up or the limit of memory is reached (as defined by the MEMLOC parameter).

14. If there is not enough memory GETSPACE branches to NOMEM (step 22). Otherwise, GETSPACE branches to OLAY (step 16).

15. The request is for overlayable memory. A loop is started to look at memory areas for an area to overlay. The start of the loop is the left off pointer, which is contained in MEMORY[BOTTOM+1].SIZEF. The upper limit of memory is the base of memory. The loop is constructed with a second pass which is used when the base of memory is found. This second pass will start the loop over at the top of memory and go down to the left off pointer. Thus, one entire pass on memory may be made.

    The loop functions as it did in the save case, however, it may be restarted if the priority of the area is greater than this stacks memory priority.

    If not enough memory is found at the end of the loop a branch is made to NOMEM (step 22).

16. GETSPACE is now at olay code. A loop is started to look at each area. For each area the following is done.

    a. If the area is in use the overlay control field of the LINKB word is set to 1 and the area is marked save.

    b. WSJUDGE is called to update the memory integral for the stack that is losing the area.

    c. If the area was not in use it will be delinked from the memory chains and marked not usable.

17. MEMLOCK is unlocked.

18. GETSPACESEARCH is called to do stack searching. Because a memory area is to be moved or overlayed, any and all descriptors that point to the area must be updated.

    GETSPACESEARCH will search for and update descriptors. It will not do the I/O to write the memory area out to disk. This is done by GETSPACE. Before we talk about how each area is to be handled some general comments about stack searching will be made.

    When a stack search is performed, GETSPACESEARCH must search any stack that could have copy descriptors. This information is available in the stacksearch graph.

For example, if the MOM is in a data base stack, all stacks linked to the data base stack must be searched. If the MOM is in a library, all stacks linked to the library must be searched. If the MOM is in the MCP all stacks must be searched.

Segment descriptors will never have copy descriptors so there is no need to search for them.

Only the process family (stacks which are in the stacksearch graph) is actually affected. While searching is in process the family is BLOCKED. Thus, GEORGE will never move to the stack.

For each area that has been selected one of the following steps will be performed. The first step that can be performed will be performed. After that step is performed GETSPACESEARCH moves on the the next area.

a. If the area is available no processing is performed for this area. The overlay control field is set to a 2.

b. GETSPACESEARCH will check the overlay control field for changes. GETSPACE has set the overlay control field to a 1. If the owner of the area releases the area, FORGETSPACE will change it to a 2 and exit. If GETSPACESEARCH finds an area in which the overlay control field has a value of 2, it will not search for copies or try to move data because the owner no longer wants the memory area.

c. If the area is sticky the MOM and all copies will point to the new area (a stack search is performed). The data is moved to the new area. The overlay control field is set to a 2.

d. GETSPACE is called to find an area large enough handle this area. On this call to GETSPACE, it will search the memory chain but will not overlay memory. If an area can be found a CORE to CORE overlay is performed. The MOM and all copies will point to the new area (a stack search is performed). The data is moved to the new area. The overlay control field is set to a 2.

e. If the data is from the code file the copies will point to the MOM (a stack search is performed). The MOM will point to the disk address where the data can be found. Because the data is from the code file the MOM can be changed directly to this address. It is not necessary to do an I/O. A CORE to LIMBO overlay has been performed. The overlay control field is set to a 2.

f. The data must be written to the overlay file. The copies will point to the MOM (a stack search is performed). The MOM is locked (all bits are turned on in the ADDRESSF). The MOM will remain locked until after the I/O is done to write the data to disk.

19. A loop is started to find any areas which require an I/O (the overlay control field is still a 1). An I/O is done to write the area to the overlay file. A CORE to OLAY overlay has been performed. After I/O operation, the MOM descriptor is updated to point to disk. That is, it is updated and unlocked.

20. GETSPACE will try to pick up the largest area. If the area below this area is available, it is delinked from the memory chains and added to the new area. If the area above this area is available, it is delinked from the memory chains and added to the new area.

21. GETSPACE branches to step 7 item e.

22. The code that follows is NOMEM code. That is, there is no memory even with the overlay. GETSPACE may:

    a. Return without memory (ICANDOWITHOUTIT).

    b. Do a DEFUNCT call (no option set).

    c. If the task is a swapjob swap it out.

    d. Wait (event MEMFREE) for memory (IWONTTELLIFYOUWONTTELL).

OUTLINE OF 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.

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.

FORGETSPACE links this consolidation maximum size available area into the correct available lists. See figure 7-4.

1. The box number and base of memory is computed.

2. The MEMLOCK word is buzzed if not locked by the caller (GETSPACE).

3. Links are checked for correct TAG values. If the links are not correct, FORGETSPACE will exit and not release the area.

4. If the overlay control field is greater than 0 (another stack wants to overlay this area) it is set to 2 (to indicate the area is available). FORGETSPACE unlocks MEMLOCK and exits.

5. If there is overlay disk reserved it is released.

6. WSJUDGE is called to update the memory integral for the stack that owns the area.

7. Do accounting to keep track of available memory.

8. If the area is in a mod that is being saved, it is added to the area accumulated so far. FORGETSPACE exits.

9. Get the largest area. If the area above this area is not inuse and usable, it is delinked from memory chains and added to the area. If the area below this area is not inuse and usable, it is delinked from memory chains and added to the area.

10. The area is linked into the chains (see figure 7-4).

11. If MEMLOCK was locked, it is unlocked. FORGETSPACE exits.

OUTLINE OF PRESENCEBIT

PRESENCEBIT is passed the P2 parameter from HARDWAREINTERRUPT. This parameter may be a data descriptor or segment descriptor. It is also passed a parameter which can be the RCW where the presence-bit occurred or information on how the P-bit is to be done. This is used for direct MCP calls.

If bit 18 is on in the address field of a data descriptor the address is relative to the beginning of a code file. The address will usually point to a value array or data pool. If bit 19 is on in the address field of a data descriptor the address is relative to the beginning of the stack's overlay file. The address field (all 20 bits) in segment descriptors is always relative to the base of a code file.

PRESENCEBIT returns a present copy descriptor with the correct memory address.

1. DESCRIPTORCHANGELOCK is locked and the define LOCKMOM is invoked. This define will see if the MOM is currently locked. If it is locked, DESCRIPTORCHANGELOCK is unlocked and LOCKMOM waits .05 seconds. When the time is

up the MOM is checked again. The process of checking the MOM will prevent two stacks from making the same array present.

2. If the address of the MOM is in the memory range of the MCPSTACK, the owner of the MOM is set to MCPSTACK. Otherwise, the stack number that owns the MOM is computed. This is found by masksearching from the address of the MOM for a tag 7 word (PCW or LINKB memory link). The stack number comes from the TAG 7 word (the stack could be doing a Pbit on an array it does not own).

3. The mom is checked to see if it is present. If so (another stack made it present) a branch is made to SAUL (Search And Unlock).

4. The mom is locked (all bits on in address field).

5. Start building option parameter for GETSPACE. User Pbits set ICANDOWITHOUTITF, MCP calls set IWONTTELLIFYOUWONTTELLF. If the call was a direct call the options are set from the second parameter.

6. If the descriptor passed is a segment descriptor the following is done.

   a. The number of words required is computed from the length of the descriptor.

   b. Information is added to the option word for GETSPACE. Because the Pbit is on code the area is readonly, no mixed tags in the area and no copy descriptors. The area is to be save if a bit is set in the segment descriptor.

7. If the descriptor passed is a data descriptor the following is done.

   a. The length of the memory area required is computed. This is based on the length field and size field in the descriptor.

   b. If the descriptor is segmented, a SEGDOPE will be set up. A SEGDOPE is save and each segment is 256 words long. The length of the SEGDOPE is computed from the length of the descriptor.

   c. If the descriptor represents a direct I/O buffer, the direct size is added to the number of words required.

   d. If the word behind the TAG 7 word found above looks (the word has a TAG of 6 and a TAG6TYPEF of 3) like a memory link, this may be a SEGDOPE. If it is a SEGDOPE, the usage of this area is set to SEGSEG

(the Pbit is on a segment of a segmented array).

e. If the mom has a 1 in the address field, the request will be for save memory.

f. If the area is overlayable, a check is made to be sure it will fit in one overlay row. If the area will not fit the stack is DSED.

g. 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 allocate space for the tags (tag transfer will be done). Users will wait for overlay disk, the MCP will not wait.

8. GETSPACE is called to get the memory. If GETSPACE returns a zero (user could not get memory, MCP would wait in GETSPACE), the following is done.

a. The MOM is unlocked.

b. If the stack is swapable it is swapped out for more memory.

c. If memory was not allocated because memory priority stopped it, the stack waits the max of .15 seconds and system factor 4.

d. If the NOMEM bit is set, the stack will wait for memory to free. Otherwise, The NOMEM bit is set and a message is displayed. The NOMEM bit will prevent a recursive Pbit.

e. Branch back to the top of PRESENCEBIT and start over.

9. If the descriptor is a data descriptor the following is done.

a. If this is a segmented array, the dope vector must be set up. The dope vector is initialized with words to represent each row. If the segmented array is from the code file (segmented value array), the address of each element must be given the proper disk address. It takes 9 sectors for each 256 word segment of the array. The compiler will start each segment on a sector boundry. After the dope vector is set up, a branch is made to SAUL.

b. If this is first time allocation of an array (address field less than 2), the area is zeroed. A branch is made to SAUL.

10. If the mom points to data in the code file (bit 18 is on) or data in the overlay file (bit 19 is on) the following is done.

    a. If the data is from the overlay file, the MOMs overlay file will be used.

    b. If the MOM stack is a segment dictionary, the code file header of the segment dictionary stack will be used when the I/O is done.

    c. If the MOM stack is not a segment dictionary stack, the proper code file must be found. The reader should note the Pbit is on something that is in the code file but has a MOM in the process stack such as file data pools.

       Unwind MOM to VECTOR to get into a stack (get out of any dope vectors).

       Masksearch from the MOM down for a FUNNY SIRW (generated by compiler). If the SIRW is found it will point to the proper segment dictionary stack. Otherwise, the segment dictionary stack for the stack that owns the MOM is used. Once the segment dictionary is found the code file header is known.

    d. If this box can not see the unit and the address of the area is not in global, a buffer is allocated in global and the I/O is done. The data will be copied to the local area and the area in global is released.

    e. If the box can see the unit it will do the I/O.

    f. If the I/O was on the MCP code file and more than one header is in use, the headers are rotated.

    g. If there was an I/O error on user code or data the stack is DSED.

    h. A branch is made to SAUL.

11. If the area is a direct I/O area, the IOCB part is initialized.

12. Label SAUL. If the MOM is present (another stack made it present) and the MOM stack is not this stack (and not a B7800 or B7900), a full stack search is made to update all copies to point to the area. Otherwise, only this stack is searched for copy descriptors. The copies are updated to point to the area.

13. If the area was made present by this stack, the mom is stored with the proper address. The RCW of the caller is

stored in the LINKC word. If the area is to be overlayable the link words are changed. GETSPACE will leave the area save so it is not overlayed before the MOM is built.

14. PRESENCEBIT returns a copy descriptor with the proper address field.

OUTLINE OF BLOCKEXIT

This routine is called at the end of each block that declared an item that allocates storage. The call to BLOCKEXIT is the last item the procedure will do before the exit. BLOCKEXIT will clean the block.

1. An array is built to point to the base of the block. The descriptor is generated using MSCW linkage.

2. A masksearch is done for the TAG 6 (SCW) word, this word will tell BLOCKEXIT what to look for. If the word is not found BLOCKEXIT exits.

3. If only part of the block is to be cleaned (NEWP cheap blocks), the array is adjusted.

4. If no bits are on in the SCW, some default bits are set. The SCW was generated by an old code file.

5. If the SCW indicated a format buffer is locked, it is unlocked.

6. If there is an EPILOG procedure, it is invoked. Care is taken not to let the user get control and branch around BLOCKEXIT.

7. If the user is exiting the critical block and there are offspring still running, it is DSED for CRITICAL BLOCK EXIT.

8. If there are libraries in the block they are disconected. If this is the last user of a temporary library it will be resumed.

9. Any interrupts linked are delinked.

10. AIT entries are cut back.

11. A loop is started to masksearch the block looking for MOM data descriptors. If they are found the following is done.

    a. If the area is present and a space (not an area), the usage of the area is used to decide how to

handle it.

FIBS are closed, dope vectors are handled by FORGETDOPEVECTORS, direct I/O buffers are handled by DIRECTOR, SIBS are handled by DMSCLOSE PIBS are destroyed. FORGETDOPEVECTORS is a recursive routine. DIRECTOR will cool off any I/O operations.

FORGETSPACE is called to release the memory.

b. If the area is an area, FORGETAREA is called.

c. If the area is non present, REMOVENONPRESENTARRAY is called. This procedure will make sure the area is not in a state of change and release the overlay disk.

12. BLOCKEXIT exits, the block is clean.

OUTLINE OF AMNESIA
------- -- -------

This procedure will release a memory area. It differs from FORGETSPACE in that it will look at the usage of the area. It will take the same action as BLOCKEXIT for each type of area.

OUTLINE OF GETAREA

GETAREA is passed one parameter, which is the size of the area required. A bit is set in the parameter if the caller can wait for the area if it is not available. The lock GETFORGET is used to protect the links.

1. Check parameter to see if requested size is too large or too small.

2. If there are no empty rows (rows that have no areas allocated in them) and the caller can wait and the caller is not AREAMANAGER, the stack will wait.

3. A link list lookup is done to search for an area that is large enough. If no area is found and the stack can not wait a fatal dump is taken. Otherwise, the stack will wait as follows.

   a. If the AREAMANAGER (an IR) is asleep it will be RESURRECTED to get more areas.

   b. Wait on the event MOREAREA (the AREAMANAGER will cause the event when more areas are available).

   c. Go back to step 3.

4. Extract the area from the list.

5. If the area is the size of a row, a new row was started. If the number of empty rows is less than the number that should be kept, AREAMANAGER is told to get more areas.

6. The excess words are returned to the list.

7. The link word is constructed.

8. The area is zeroed and the address is returned to the caller.

OUTLINE OF FORGETAREA

FORGETAREA is passed one parameter, which contains the address of the area.

1. The area is placed in the proper place in the row and links are updated.

2. If the return of this area has created an empty row and there are too many empty rows, this row is returned

(FORGETSPACE is called). Otherwise, the number of empty rows is incremented.

OUTLINE OF FINDOLAYSPACE

FINDOLAYSPACE is called to find overlay disk for an array. The OLAYINFOLOCK is used to protect overlay disk structures. The four parameters passed are as follows.

The stack number of the owner of the array.

The descriptor the overlay disk is required for.

The number of segments required.

A boolean to indicate if the stack can wait for overlay disk.

1. If the number of segments required is zero, it is computed using the descriptor passed.

2. For each row in the overlay file a procedure (TAKE) is called to try to find the proper number of segments. If the segments are found the following is done.

   a. The amount of virtual memory in use is updated.

   b. The disk address (record number) is returned to the caller.

3. If the disk was not found, the NEEDAROW bit is set in the STACKINFO array. The event OLAYSPACEWANTED is caused.

4. If the stack can wait for overlay disk the following is done.

   a. FORK OLAYSCOUT and wait for OLAYSPACEGOTTEN.

   b. When the event is caused the stack will see if its NEEDAROW bit is still on. If the bit is still on it will continue to wait.

   c. OLAYSCOUT will get disk for any stack with the NEEDAROW bit on. After it gets disk it will turn the bit off and cause OLAYSPACEGOTTEN. Once the stack has disk it will try to find disk again.

5. If the stack could not wait, it will FORK OLAYSCOUT and return a zero.

OUTLINE OF LOSEOLAYSPACE

LOSEOLAYSPACE is called to return overlay disk to the overlay file. The procedure is passed four parameters as follows.

The stack number of the owner of the array.

The descriptor that represents the area.

The disk address (record number) in the overlay file.

The number of segments being returned.

1. If the disk address passed is zero, it is set up using the descriptor passed.

2. If the number of segments passed is zero, it is set up using the descriptor passed in.

3. The amount of virtual memory in use is updated.

4. The disk is placed back in the overlay disk file (bits in OLAYINFO array are turned off).

OUTLINE OF WSSHERIFF

WSSHERIFF will do non demand overlay. That is, it will try to keep memory clear of old arrays. This procedure (an IR) is forked any time the overlay goal is non zero. There is one copy of WSSHERIFF in each box.

1. WSSHERIFF will keep track of how long it takes to run through one cycle, any time that is left over it will wait. The cycle time (how often it runs) is 3 seconds.

2. An endtime is computed, which is when the overlay process will stop even if we are not done. It is set to cycle time from now.

3. If the overlay goal is zero and no stacks have been suspended WSSHERIFF will go to end of task. If a stack has been suspended WSSHERIFF will hang around long enough to wake it up.

4. If there are suspended tasks and there is enough memory or we must wake up a stack because the overlay goal is zero, the following is done.

   a. The stack with the best priority, oldest and will fit into memory will be allowed to run.

   b. Once a stack has been found the REPLYEVENT for the

stack is caused.

5. If the overaly goal is zero or someone is trying to stop overlay, we wait.

6. Overlay goals for each stack are derived as follows.

   a. The conversion factor % memory per min to % memory per cycle is made.

   b. The COREINUSE-SAVECOREINUSE for a stack is the overlayable memory the stack is using.

   c. Each stack in the box has an overlay amount computed. The overlay amount is the overlayable memory multiplied by the overlay rate. This overlay amount is stored in a word in the PIB.

   d. A total amount of memory to be overlayed is also computed.

7. Look at segments in memory starting at the left off pointer and going down thru memory. If the end of memory is hit, start at the top. Care is taken to be sure only one pass on memory is made in a cycle.

8. For each segment that is in use, not save, belongs to a stack that has a non zero overlay amount and has an address in the overlay file (or belongs in the code file) the following is done.

   a. The links are marked as save and involved in an overlay.

   b. The family that owns the area is blocked from running. This will allow a search of the family.

   c. The overlay amount for the stack is decremented.

   d. The memory integral is updated.

   e. Search for copy descriptors and make them point to the MOM. There is no need to search for copy descriptors if the MOM is a segment descriptor.

   f. If the area is from the code file the CORE to LIMBO counts are updated. The mom will point to the code file and the family is unblocked.

   g. If the area is from the overlay file the mom is locked and the family is unblocked. The CORE to OVERLAY counts are updated. An I/O is done to write out the area. The mom is then updated to point to the overlay file.

h. FORGETSPACE is called to release the area.

9. If there is still more memory to be overlayed, branch back to look at another segment.

10. If no more memory is to be overlayed, check to see if a stack should be suspended. If the amount of available memory is below the minimum a stack will suspended as follows.

    a. The stack to be stopped must not be sorting and must have a normal HISTORY word. The lowest priority stack is stopped. If two stacks are of equal priority the stack that has been running the shortest time is stopped.

    b. KANGAROO is called to stop the stack.

11. Branch back to the wait statement.

SECTION 8

JOB CONTROL

INTRODUCTION

Jobs are run to control the sequence of tasks. Jobs are compiled by the WORK FLOW LANGUAGE (WFL) compiler and may enter the system via a card reader, MCS or ODT. A job is defined in this text as any program compiled by the WFL compiler. After a job is compiled, it is entered into a JOB QUEUE by the CONTROLLER. When jobs end BACKUP files must be printed or punched. These aspects of jobs will be covered in this section.

Due to the wide area that JOB control covers, this section has been divided into 3 sub-sections:

1. WORK FLOW MANAGEMENT

2. CONTROLLER

3. AUTOBACKUP

The first sub-section is about the Work Flow Management (WFM) system, which includes the WFL compiler and CONTROLLER. The second sub-section explains CONTROLLER's involvement in JOB QUEUES and is followed by a discussion on AUTOBACKUP and how it relates to jobs, CONTROLLER and JOBFORMATTER.

WORK FLOW MANAGEMENT

This section provides information relating to the Work Flow Management system and is provided here for completeness. Much of the same information can be found in section 1 of the Operator Display Terminal (ODT) manual (form number 5011687).

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 Management system was to localize those functions relating to work flow control. 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 MCP includes five separately compiled program units: MCPHOST, CONTROLLER, WORK FLOW LANGUAGE (WFL) COMPILER, ALGOLSUPPLEMENT and JOBFORMATTER. All of these program units are written in DCALGOL except the MCPHOST which is written in NEWP. A complete MCP is generated by binding the DCALGOL units into the MCPHOST.

The CONTROLLER manages the display routines and the scheduling queues. 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. ALGOLSUPPLEMENT contains routines used for system maintenance.

Figure 8-1 shows how these separate units interact. There are two external influences on the system: job decks and operator input. Subsequent sections document how each is handled by the system.

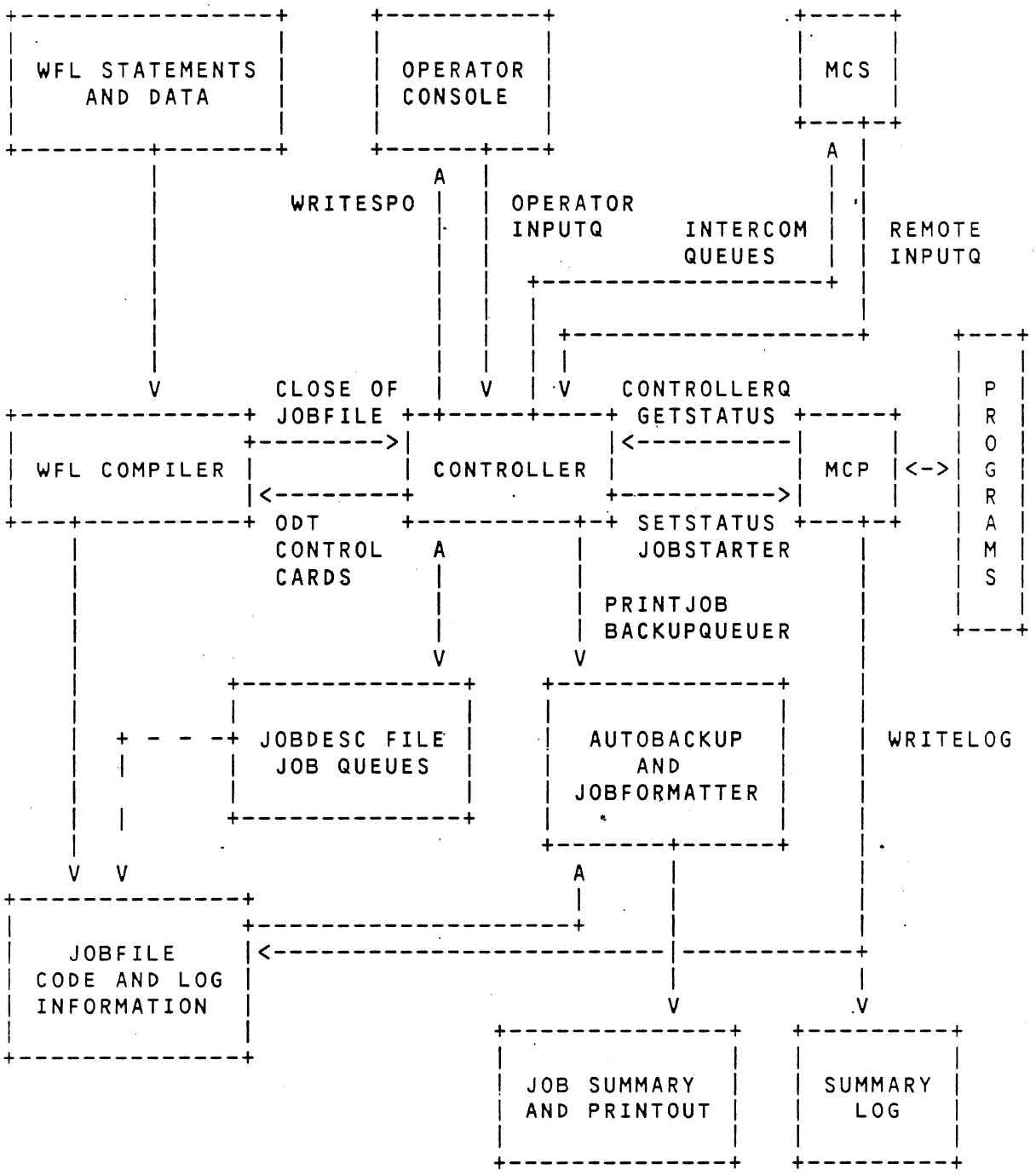


Figure 8-1. WFM System Organization

WFL COMPILER

When a job deck is placed in the card reader, the ETERNALIR procedure of the MCP will note that the reader has changed status (gone from not ready to ready). ETERNALIR will initiate the WFL compiler as an invisible independent runner. The WFL compiler can also be programmatically invoked.

The WFL compiler can be called to compile a job or handle assignment to a tasks FILECARDS (TASKID.FILECARDS). In addition, WFL is responsible for FA requests. Contained in the WFL symbolic are the 3 separate procedures CCSTRINGCONV, CCSTRINGFUNCTION and 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).

If the WFL compiler was invoked to compile a job, it 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. Into this file it places object code to run the tasks of that job, copies of the control cards (for later printout), data decks belonging to the job, space for logging and restart information. Once compiled, a jobfile is a self-sufficient representation of a job. The WFL compiler will not go to end of job until all jobs placed in the card reader have been compiled. As each jobfile is completed, the disk file is locked. When the last job has been compiled, WFL leaves the mix.

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.

Figure 8-2 is that of a typical JOBFILE. Within segment zero of a job's code file are pointers to:

The logging space.

The job's PPB.

The segment dictionary.

The process stack.

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.

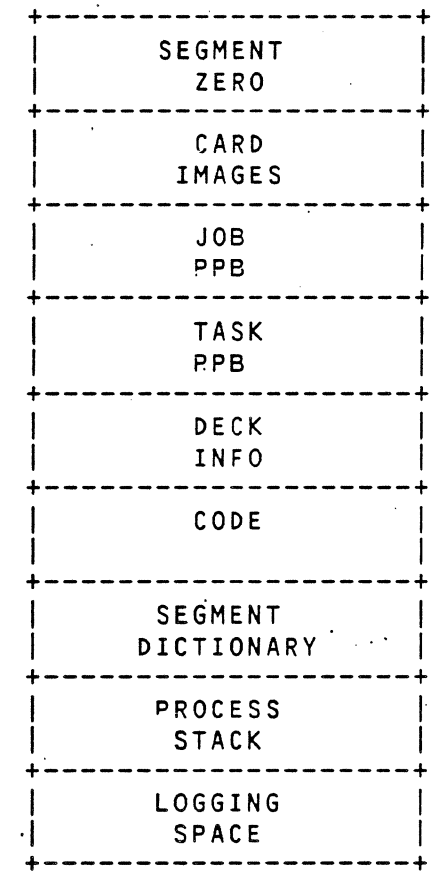


Figure 8-2. Typical JOBFILE

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. Responding to the SCHEDULEREQUEST, CONTROLLER will place an entry for the new job in a JOB QUEUE. 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, 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 8-1 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 and has no resource restriction statements, it is put into the default queue.

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 (RESOURCE statement), lines printed and cards punched. For example, if a job has 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 8-3 for a more complete version of the queue-matching algorithm.

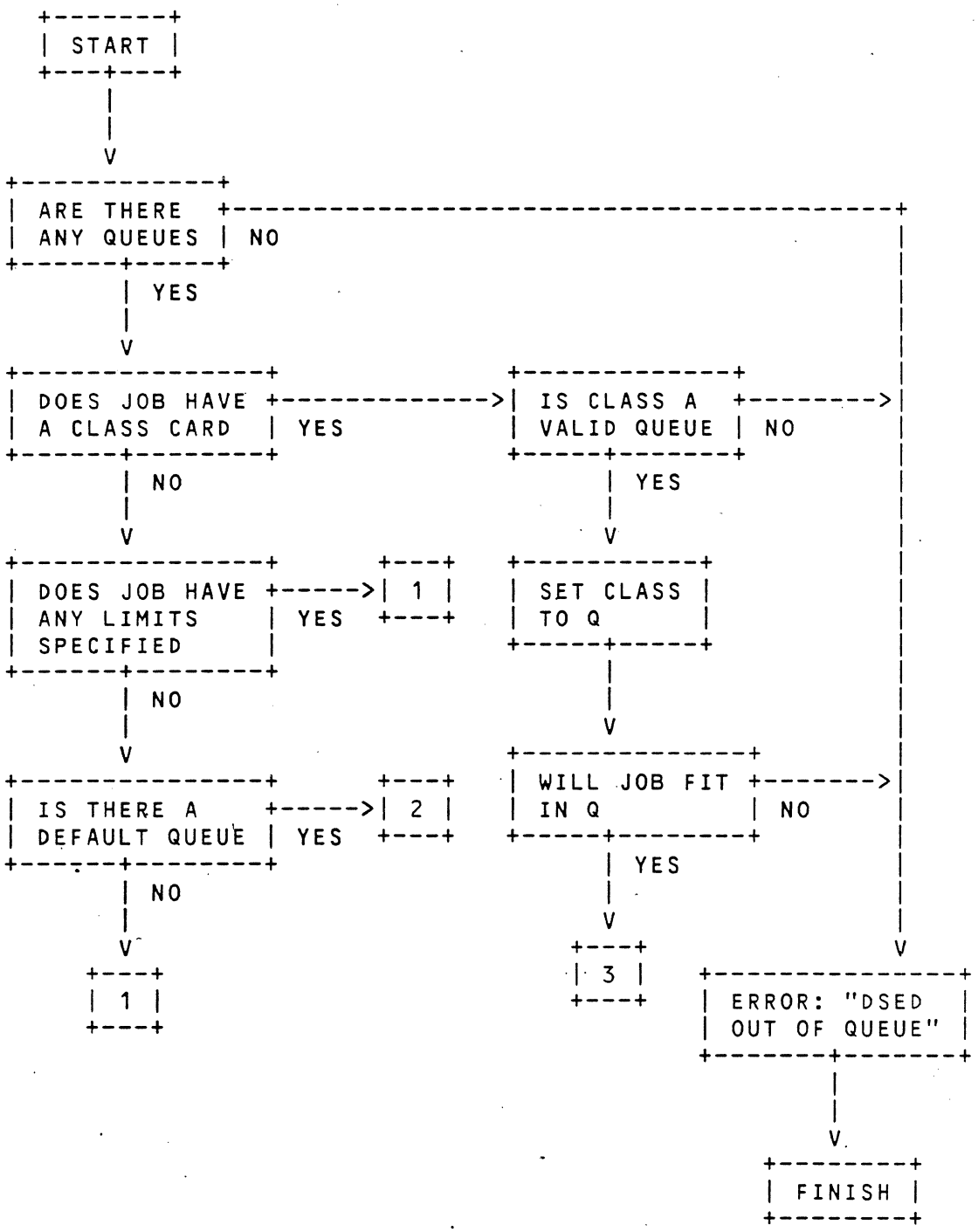


Figure 8-3. Job Enqueuing Algorithm (page 1 of 2)

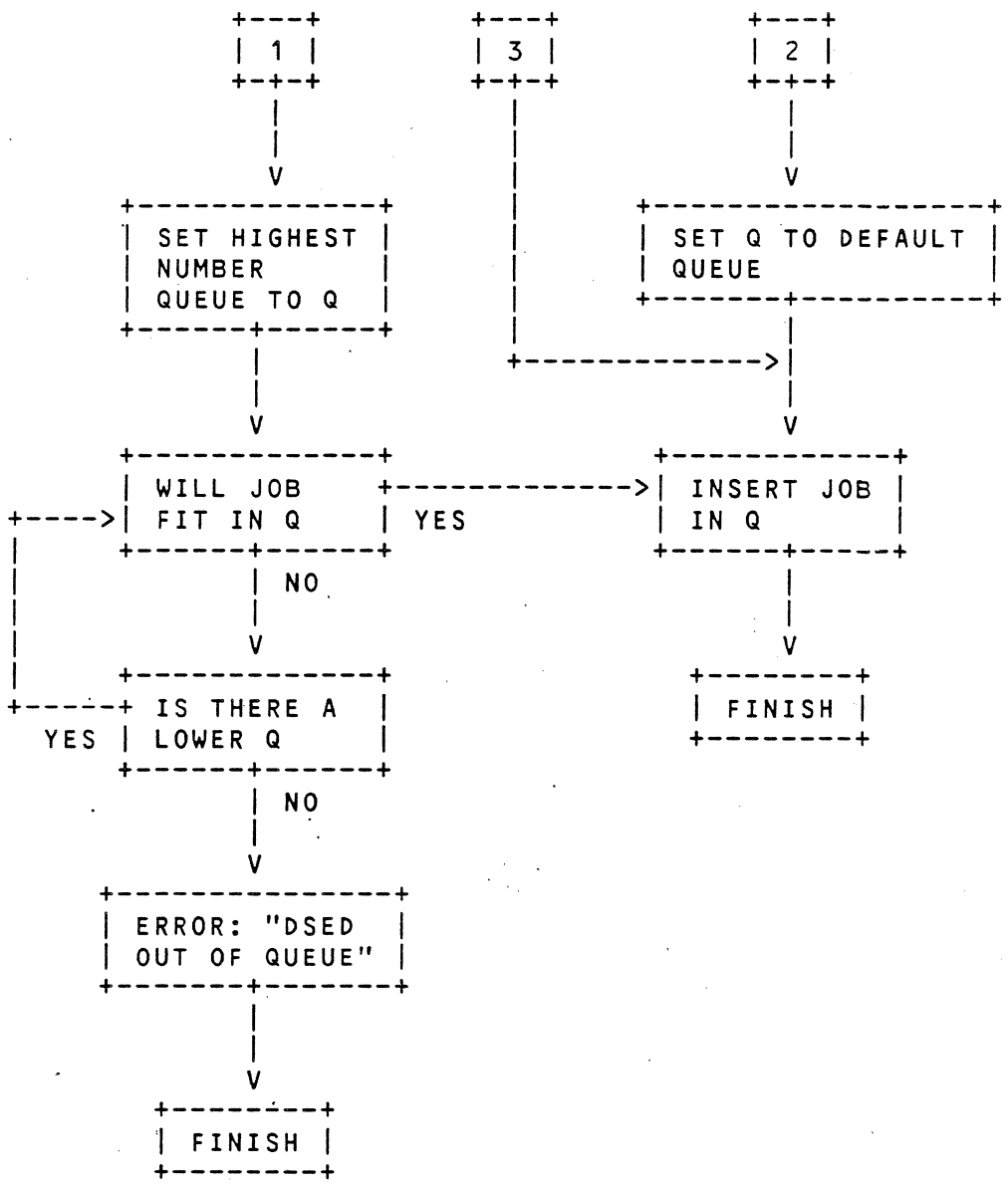


Figure 8-3. Job Enqueuing Algorithm (page 2 of 2)

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 priority of a job may not be placed 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 and initiates the task via DOCTOR.

While the job is active (or scheduled at the MCP level), the jobfile is used in the following ways:

1. The jobfile is the code file for job control statements, so that it is referenced, as any code file, to load executable code into memory.

2. The jobfile contains the card data input needed for the job plus pointers to that data.

3. The jobfile 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 8-1).

4. 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. Thus, jobfiles are queued for printing, just as any BD file would be.

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. GETSTATUS returns answers to any interrogations made by the operator; SETSTATUS 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, OPERATORINPUTQ, receives ODT inputs. These messages are placed in the OPERATORINPUTQ by KEYIN (an independent runner). 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. The DCALGOL procedure WRITESPO is used to display information on the ODT. 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.

If the operator types in a job control deck at the console 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.

A log entry type (major type 7) 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.

CONTROLLER

The CONTROLLER has 3 major responsibilities, which are:

1. Absolute control of JOB QUEUES.

2. ODT screen displays (ADM).

3. Executing most operator messages (via SETSTATUS and GETSTATUS).

This sub-section is mainly concerned with CONTROLLER's JOB QUEUE responsibilities.

JOB ENQUEUEING

CONTROLLER is normally found waiting on several events. One of the events, ADMTIMEOUT, is a real procedure and produces a wait time. The other events are located in the HIDDEN MESSAGE of the associated queues.

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 are passed.

Referring to figure 8-1, closing a JOBFILE will cause an entry to be placed in 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. A SCHEDULEREQUEST code causes CONTROLLER to call its local procedure NEWENTRY.

The NEWENTRY procedure within CONTROLLER starts execution by calling the MCP procedure GETJOBDESCRIPTION (passing COREINDEX). GETJOBDESCRIPTION returns the HDR/PPB (JOBFILE header) combination which is subsequently written to the JOBDESC file at an available location. This HDR/PPB forms a JOB QUEUE ENTRY and is shown in figure 8-4.

The first word in the HDR/PPB (built from JOBFILE header) contains a field that points to the PPB. A word known as DISKLINK is 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 8-1 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.

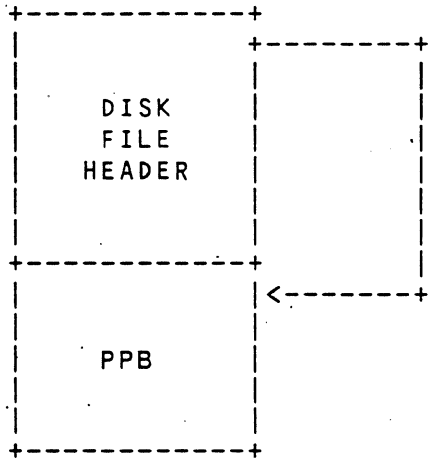


Figure 8-4. JOBDESC-JOB QUEUE ENTRY

NEWENTRY now calls another CONTROLLER procedure named ABSTRACT.

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 8-5). This array (in WINDOW format) contains the job's attributes such as PROCESSTIME, IOTIME and other job parameters. ABSTRACT returns a QUEUE INDEX, but note that QUEUE INDEX is not the same as 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 Job attributes 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 8-6 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.

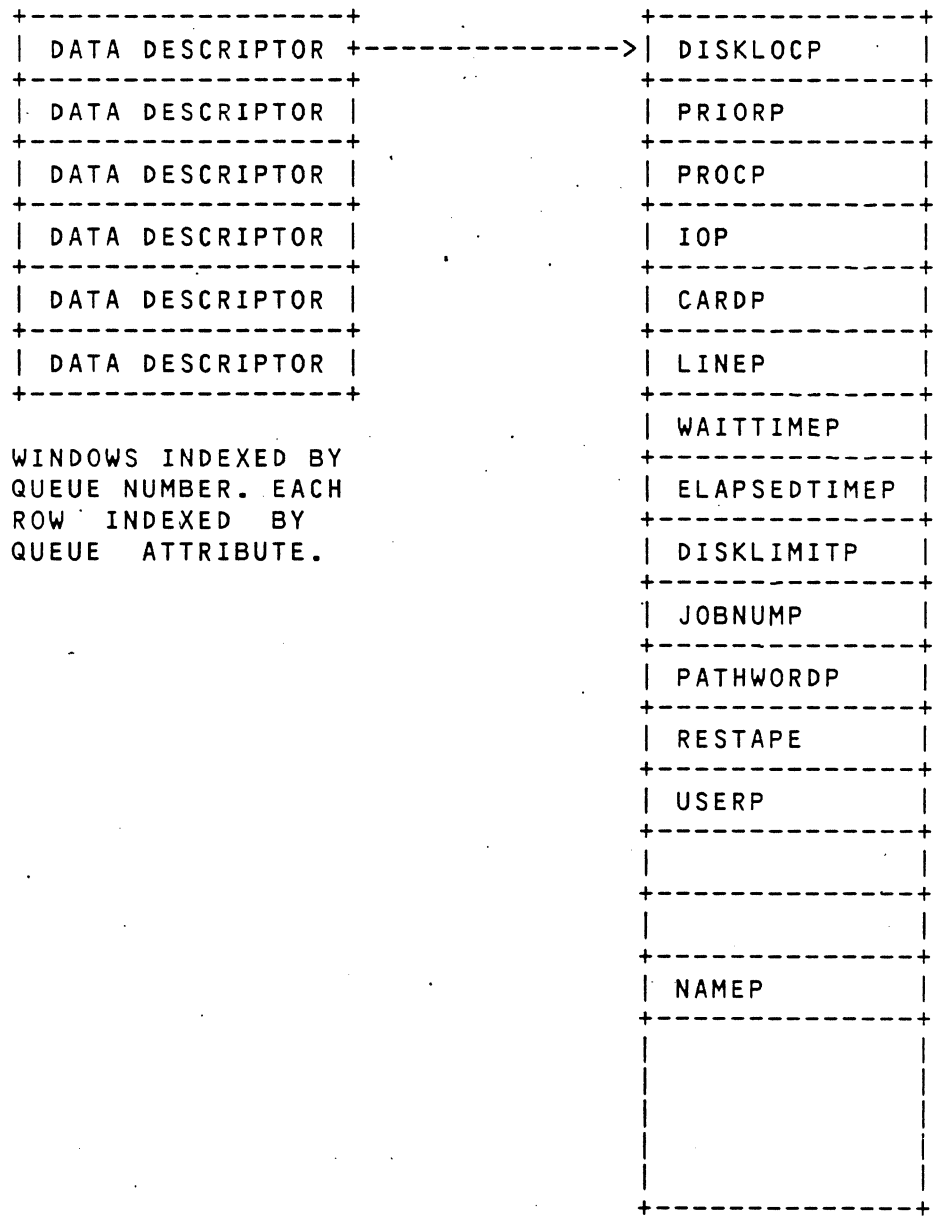


Figure 8-5. WINDOWS ARRAY.

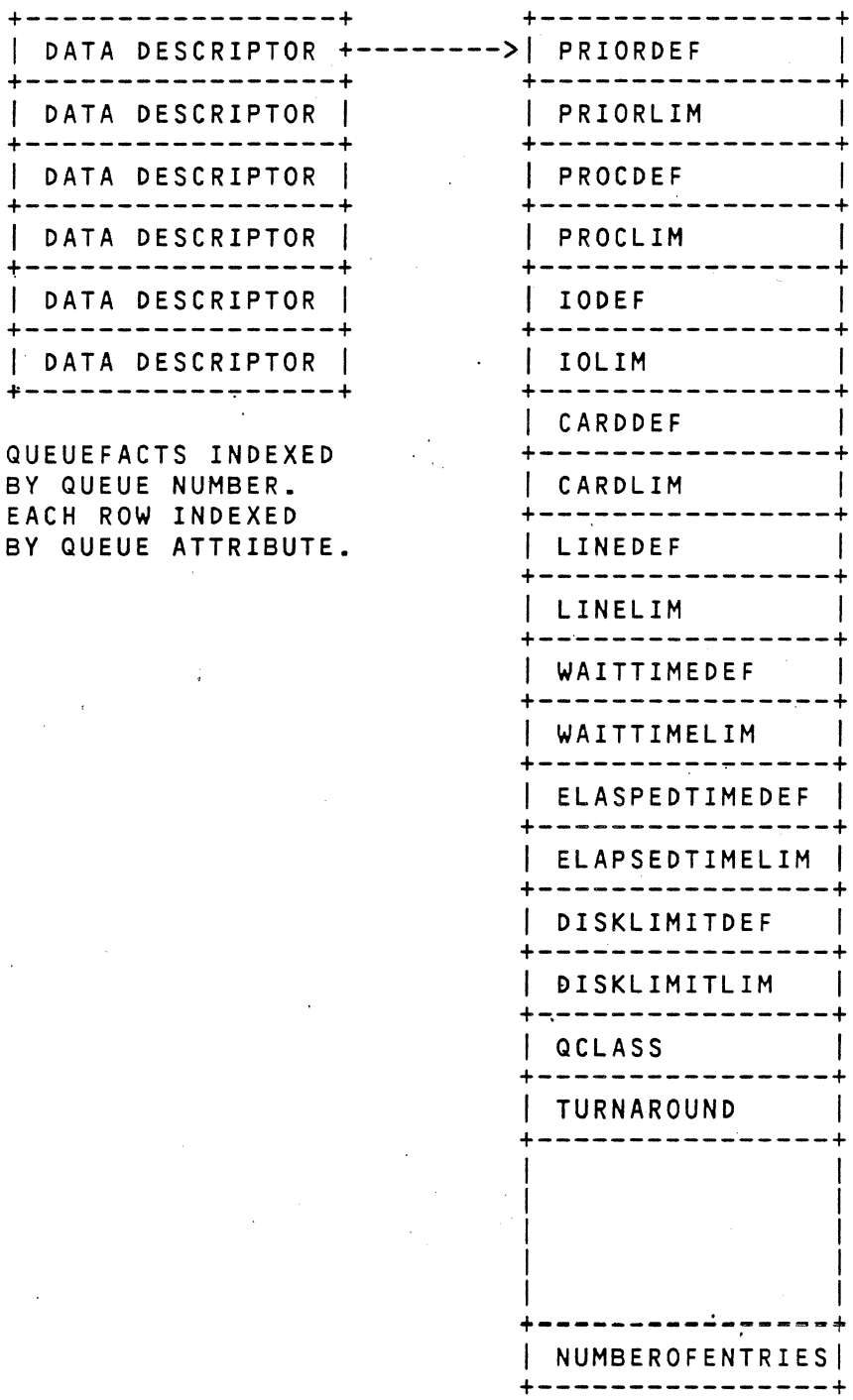


Figure 8-6. QUEUEFACTS ARRAY.

NEWENTRY now calls another CONTROLLER procedure named QUEUEINSERT.

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 8-5) the new job will be placed in the WINDOW. The job is then linked into the DISKQUEUE (Figure 8-7) of all jobs with a similar priority for the given queue. For each queue, DISKQUEUES, contains the head and tail disk addresses for all jobs linked together by similar priority. The disk link word is used to link the JOB QUEUE ENTRY (HDR/PPB) into the queue. The JOB ENQUEUING ALGORITHM is shown in figure 8-3 but will not be reviewed here.

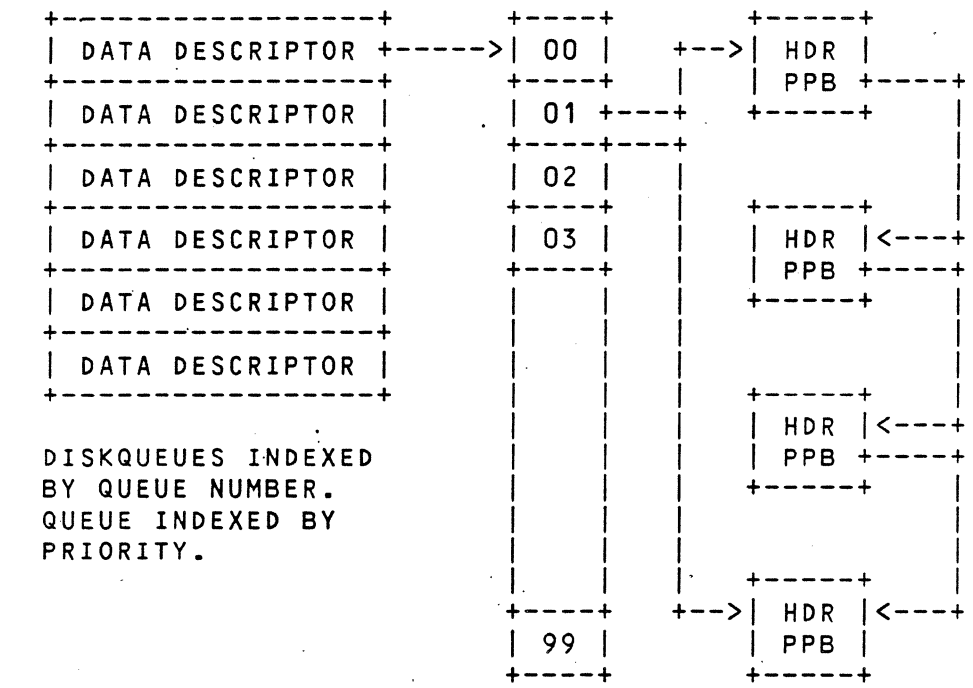


Figure 8-7. DISKQUEUES ARRAY.

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 8-8 shows a flow of the JOB DEQUEUING ALGORITHM.

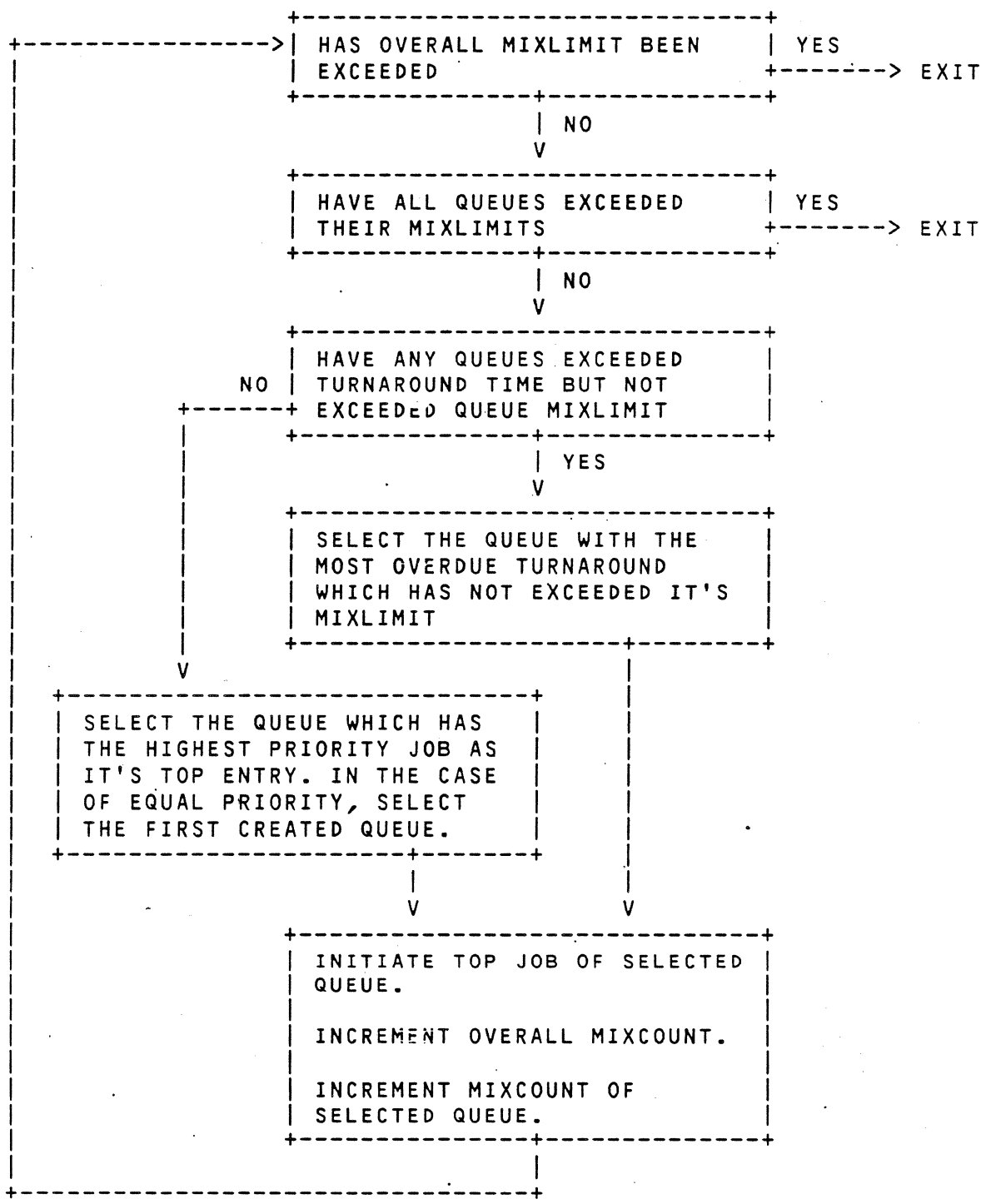


Figure 8-8. JOB DEQUEUING ALGORITHM

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
```

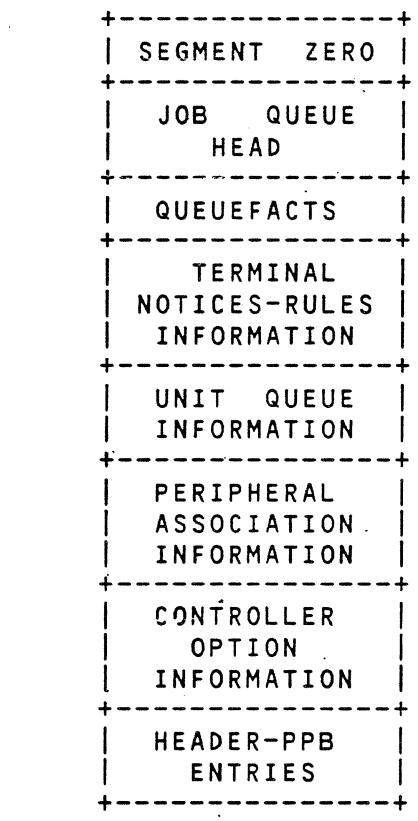
When the message is entered and the ODT placed in transmit, the hardware receives a status change and calls the MCP procedure HARDWAREINTERRUPT. This procedure, seeing that it is a status change for an ODT sends a message telling KEYIN to read the ODT.

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 8-1.

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.

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. In addition to this, the JOBDESC file is used to save CONTROLLER operation information, PERIPHERAL ASSOCIATION information and other run time information. These JOBDESC areas and others can be seen in figure 8-9. SEGMENT ZERO (0) of the JOBDESC file is of fixed format and contains pointers to the above items.



ITEMS IN JOBDESC ARE POINTED TO BY WORDS IN SEGMENT ZERO

Figure 8-9. TYPICAL JOBDESC

The following paragraphs describe the contents of each section of the JOBDESC file.

The JOB QUEUE HEAD section contains the number of queues and pointers to the queues. There is a one word pointer for each queue.

The QUEUEFACTS arrays (Figure 8-6) 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. This area contains the queue factors (attributes).

TERM controls number of lines to be displayed, width of lines, where first line should start and other terminal information.

NOTICES controls the information used to drive the ODT. It contains the state of the ADM, when a new ADM should be displayed and other information.

RULES is where the information from the ADM message is saved.

The UNIT QUEUE area in JOBDESC is used to save information from the "UQ" ODT message. This message is used to associate units with queues. For further information see "UQ" message in the ODT manual.

The PERIPHERAL ASSOCIATION 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

The header in the JOBDESC file is the same as headers found in the FLAT DIRECTORY with a few exceptions. These headers have job linkage words (for queue linkage) and backup control (job printing) information.

DISKMAP ARRAY

The "HEX" array DISKMAP 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.

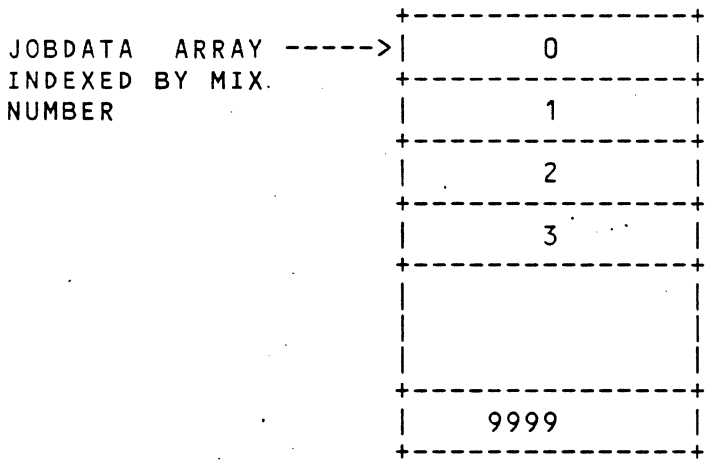
In reference to the procedure GETDISK, DISKMAP is searched for a contiguous group of 4"0" entries indicating that the associated segments are available.

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.

JOBDATA ARRAY

The JOBDATA array, Figure 8-10, is an 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 is used 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.



THE LOCATIONF FIELD (FOR ANY VALID JOB ENTRY) CONTAINS THE LOCATION IN THE JOBDESC OF THE JOB (HDR/PPB)

Figure 8-10. JOBDATA ARRAY

AUTOBACKUP

Information to be written to a printer or punched on 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 8-1, the MCP by use of the CONTROLLERQ, notifies CONTROLLER that a job has finished. It is now time 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 8-11. When a job ends it exits into NORMALEOJ like any program. NORMALEOJ, 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 BACKUPQUEUER returns a 0 which will cause AUTOBACKUP to leave the mix. If there are more entries in the queue 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.

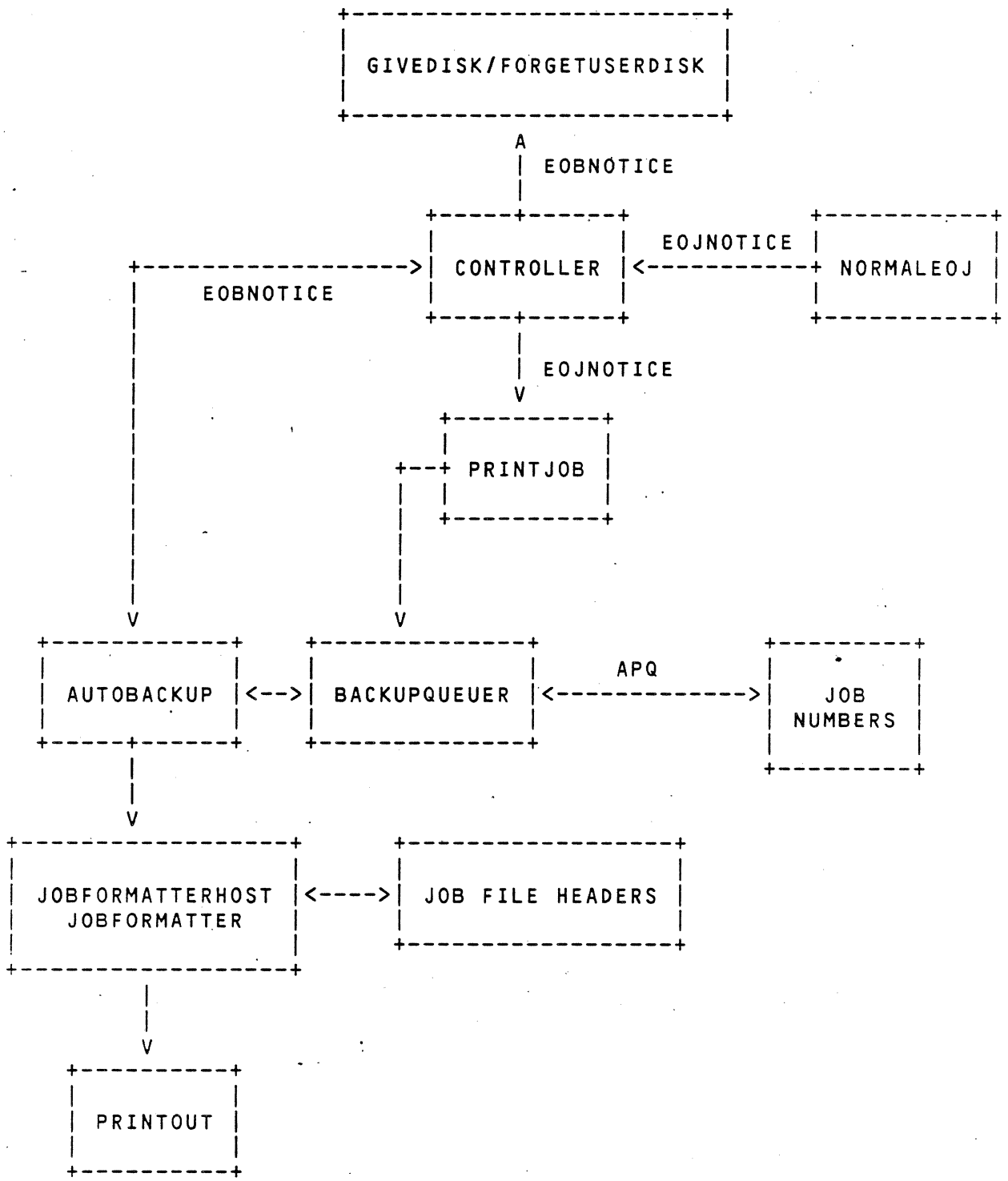


Figure 8-11. AUTOBACKUP ABSTRACT

SECTION 9

PROCESS CONTROL

INTRODUCTION

The control and initiation of programs has many aspects. Initiating a program includes assignment of task attributes, locating the code file, building the process stack, building the segment dictionary stack, bringing code into memory and executing the program. Programs may cause other programs (tasks) to be executed and affect those programs once they are running. Also, how one task 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 3 sub-sections:

1. PROGRAM INITIATION AND TERMINATION

2. EVENTS

3. PROCESS CONTROL EXAMPLE

The first sub-section, PROGRAM INITIATION AND TERMINATION, discusses how process stacks are set up, code located and programs executated. The second sub-section, EVENTS, discusses events and also provides information about specific MCP procedures such as GEORGE, WAITP, CAUSEP and TIMETUNNEL. The last sub-section is an example ALGOL program initiating tasks and using events. This sub-section is followed by several procedure outlines.

PROGRAM INITIATION AND TERMINATION

Program initiation consists of establishment of a memory area for the Program Information Block (PIB) and assignment of task attributes to the PIB. The code file must be located. Parameters (if any) must be checked and assigned to the program. A stack number and mix number must be assigned. Linkage must be established between the new task, its parent

task (if any) and siblings (if any). A memory area for the process stack is established and initialized. The stack number is placed in the READY QUEUE. When a processor becomes available it will move to the new stack and EXIT into a procedure in the MCP (NORMALBOJ). A segment dictionary stack will be set up and a BOJ log entry made. The entry point from the code file is used to enter (with an EXIT from NORMALBOJ) the users 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 release the memory area for 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). ETERNALIR (because of the TERMINATE entry) waits for the processor to leave the stack and releases the memory area for the process stack.

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 (called by DELIVERY) 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). After a PIB has been partially set up the MCP procedure, DOCTOR, is called.

DOCTOR is responsible for task attribute assignment. DOCTOR will assign attributes to the new PIB from the parent task and in some cases from the job. If a code file must be found (initiation is not on an internal procedure), EXTERNALREFERENCE of DOCTOR is called. EXTERNALREFERENCE will call FINDAFILE passing the name (MYNAME) in the PIB. FINDAFILE will locate the code file and place a descriptor to the file's header in the DISK FILE HEADER STACK. FINDAFILE will return the index (COREINDEX) into the DISK FILE HEADER STACK to EXTERNALREFERENCE. EXTERNALREFERENCE places the code file COREINDEX in the TASKINFO word in the PIB.

EXTERNALREFERENCE will also extract information from the code file and place it in the PIB. The minimum contents of any code file is a segment dictionary and object code segments. Due to various considerations, all code files have a fixed format SEGMENT 0. This disk segment is the first segment in the file and contains an entry point index (index into the segment dictionary to PCW that points to the outer block of the program), parameter information, memory estimate, segment dictionary stack image pointer, stack estimate, code file

level and a pointer to the Program Parameter Block (PPB). The Program Parameter Block contains information about the program such as priority, printlimit, process time and other program parameters. The actual layout of segment zero is documented in the COMPILERSUPPORT module of the MCP.

There are descriptors in the segment dictionary that are used to locate the object code. The code file also contains value arrays, data pools, File Parameter Blocks (FPB) and binder information.

After EXTERNALREFERENCE has assigned attributes from the code file, it will exit back to DOCTOR where additional attribute assignment takes place. DOCTOR then calls INITIATE.

INITIATE calls PICKASTACK to get a stack number and mix number. This information is placed in the SERIAL word in the PIB. Once a stack number has been obtained the new task owns an entry in STACKINFO, STACKSTATUS, PIBVECTOR and STACKVECTOR. The PIB is placed in the PIBVECTOR. The priority is computed and placed in STACKSTATUS. The STACKINFO entry is initialized. Associated tasks are linked together with the PROCESSFAMILYLINK word in the PIB. A memory area is obtained for the process stack and a descriptor to this stack is placed in the STACKVECTOR. INITIATE sets 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, 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 when its stack number is placed in the READY QUEUE for the first time.

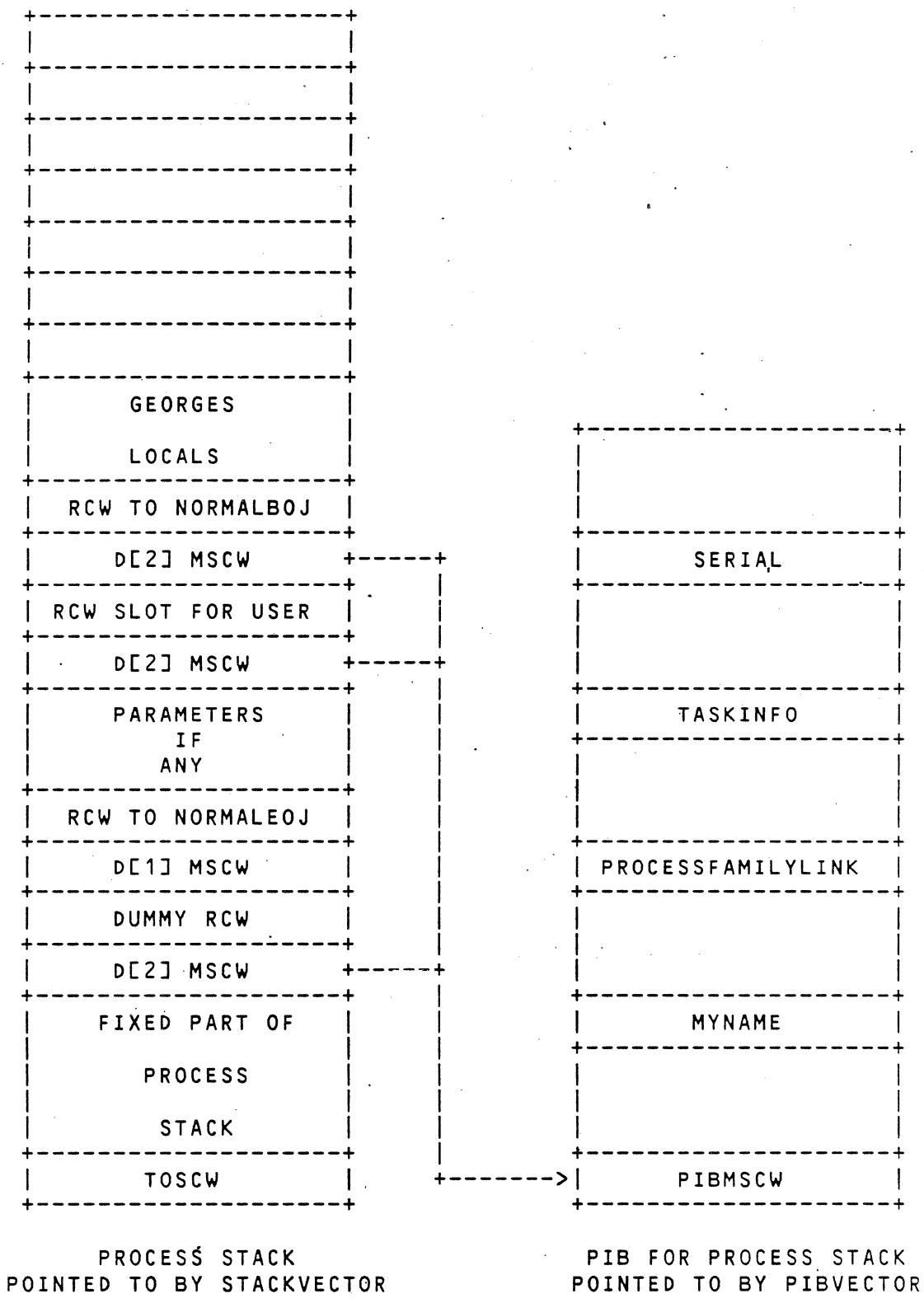


Figure 9-1. Process Stack Before First MVST Operator

The MCP procedure, GEORGE, looks at the READY QUEUE. The READY QUEUE is a linked list of stacks waiting for a processor and is maintained in priority order. The stack number of the first entry is in a MCP variable called READYQHEAD. The STACKSTATUS entry for this stack will point to the next entry. Thus, GEORGE will find the stack of best priority and move to it by executing a MVST instruction. The MVST instruction will build a TOSCW from current register settings and store the TOSCW in the first word (MEMORY[BOSR]) of the stack the processor is on. The descriptor for the new stack is fetched (from the STACKVECTOR) and used to set BOSR and LOSR. The TOSCW for the new stack is fetched (from MEMORY[BOSR]) and replaced with a processor ID. The TOSCW is used to set up S, F and a D register update is performed. GEORGE will then clock the user on the processor and EXIT into the stacks code (the RCW at MEMORY[F+1] is used) environment.

Once the new stack is the best priority stack in the READY QUEUE, GEORGE will move to it. The TOSCW is used to set up S, F and, in the example of figure 9-1, D[2]. Figure 9-2 shows the register and stack relationship after the MVST is complete.

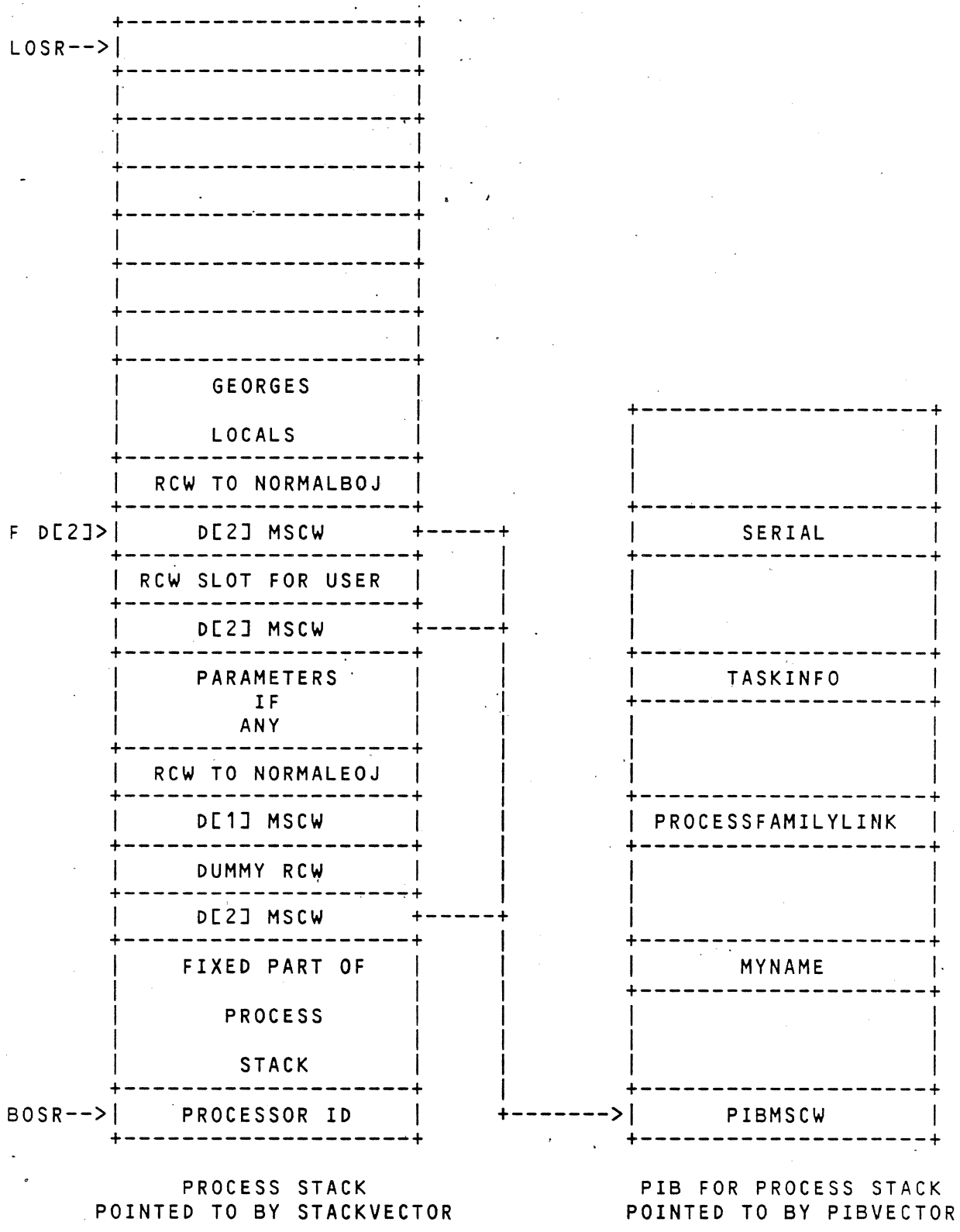


Figure 9-2. Process Stack After MVST Operator

When the EXIT instruction is executed, stack cutback and display update occur. At this time, the instruction pointers (code registers) are also changed. When GEORGE exits, it will be into NORMALBOJ.

NORMALBOJ will set up the overlay file for the stack. These structures are discussed in section 7. NORMALBOJ will check a word (D1LINK) in the header for the code file. This word contains the stack number of the segment dictionary stack if one has been set up. If a segment dictionary stack has been set up (and is visible to this box on a tightly coupled system), it will be linked to. In addition, the RUNNINGCOUNT in the PIB for the segment dictionary will be incremented. If a segment dictionary had not been set up, it will be set up at this time. The segment dictionary is read from the code file and a PIB generated. The segment dictionary is read with a tag transfer format. This format extracts the three bit tag from the first 16 bits read (4"0700" on disk is a tag value of 7). The code segments that are read as needed are read with a disk read force tag of 3. The RUNNINGCOUNT is set to one. The D[1] MSCW in the process stack is given the proper lex level for the entry point. A BOJ log record is written. The users entry point PCW is changed to a RCW and placed in the process stack. The slot is labeled USERS RCW in figure 9-3.

Figure 9-3 shows the stack 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 entry point PCW in the segment dictionary to a 3 and then writing it into the process stack at the location shown. When NORMALBOJ exits it will be into the users program.

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 PIB even though the PIB 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.

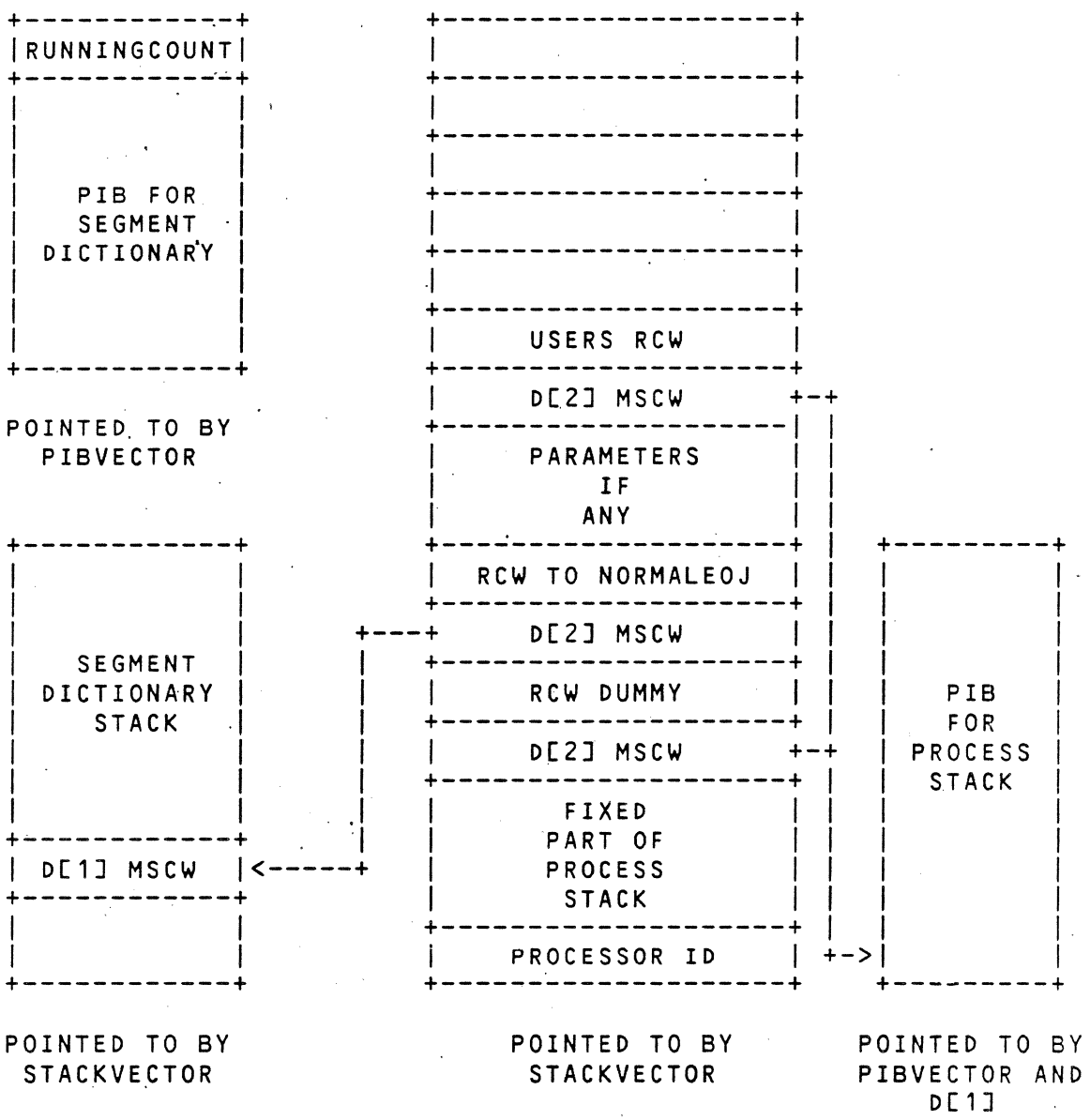


Figure 9-3. BOJ Responsibilities

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 disassociated 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.

NORMALEOJ calls WRITELOG to make an EOJ log entry. The RUNNINGCOUNT for the segment dictionary is decremented. If the RUNNINGCOUNT is zero the segment dictionary will be terminated (by TERMINATED1STACK) and the PIB for segment dictionary will be released. Arrays in the fixed base portion of the process stack are released. FORGETUSERDISK is called on the overlay header (OLAYFILEDESC) if the stack owns it's overlay file.

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 can not be released (by FORGETSPACE) at this time. In reference to figure 9-5, the following occurs:

1. Tell ETERNALIR to terminate the stack.

2. Call GEORGE to get the processor out of the stack.

ETERNALIR will get the terminate message and 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.

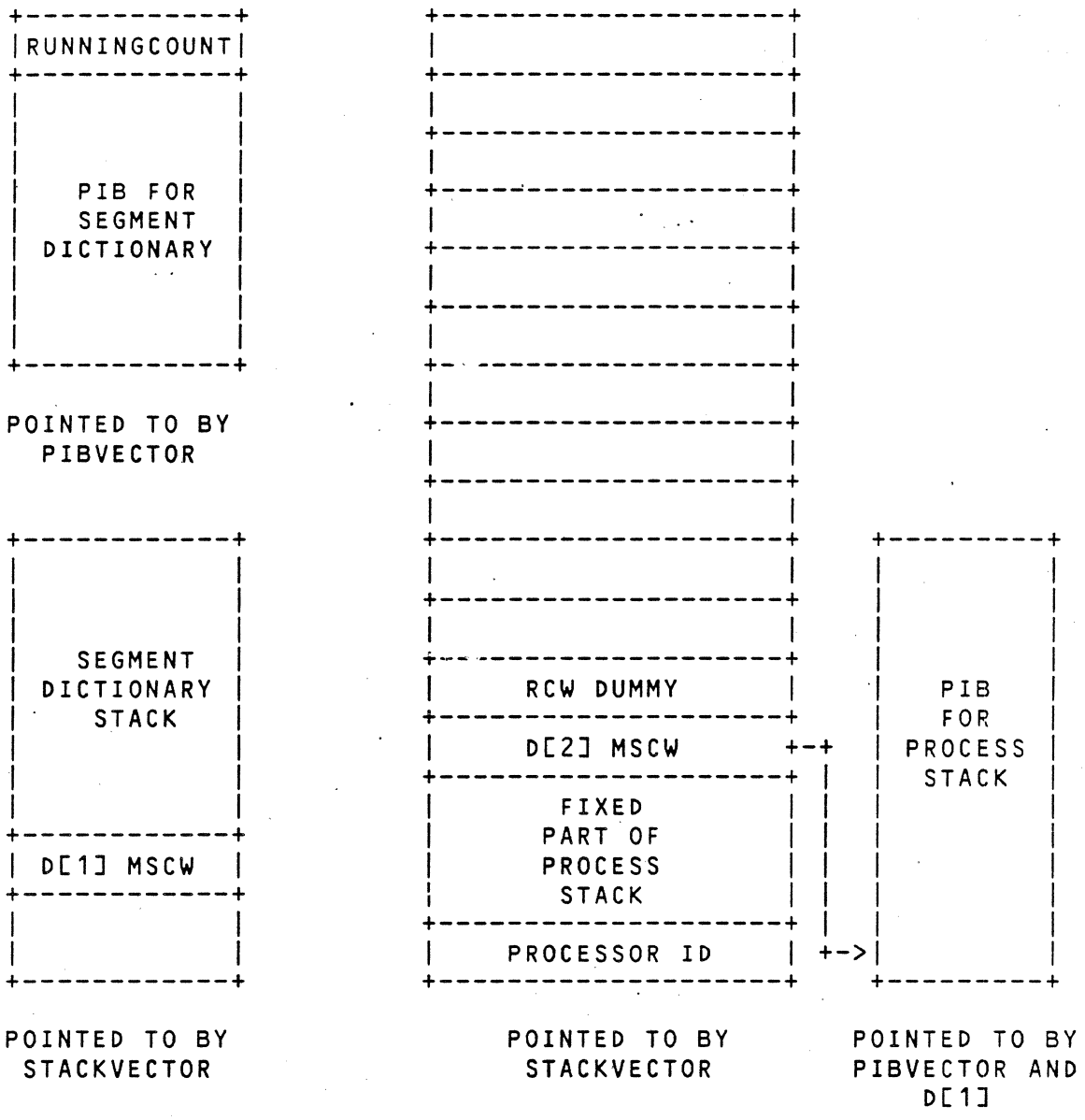
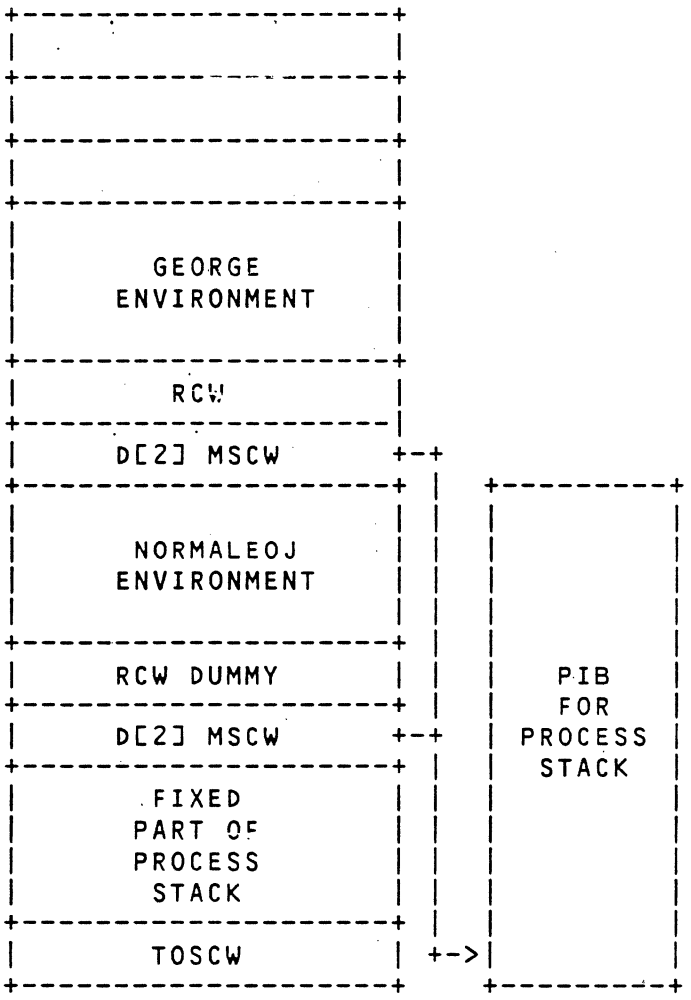


Figure 9-4. NORMALEOJ Responsibilities



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

EVENTS

This sub-section starts with a discussion on events and software interrupts. This is followed by a discussion on the MCP procedures TIMETUNNEL, WAITP and CAUSEP.

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 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 the INTERRUPT MECHANISM.

EVENT DECLARATION

The EVENT is a special type of variable declared in a manner similar to other types of variables. In ALGOL and NEWP an event, E1, would be declared as:

```
EVENT E1
```

As with other types of variables, the EVENT can be passed as a 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 condition-oriented functions is as follows:

a. SET FUNCTION:

ALGOL

```
SET (<event-ID>)
```

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:

ALGOL

```
RESET (<event-ID>)
```

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:

ALGOL NEWP

```
CAUSE (<event-ID>)
```

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 READY QUEUE. The READY QUEUE is a queue of all tasks in priority order, that are capable of running. Tasks are taken out of the READY QUEUE 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 caused. A task will only be placed into actual execution when it is the top item in the READY QUEUE and a processor is available.

d. CAUSE AND RESET FUNCTION:

ALGOL NEWP

CAUSEANDRESET (<event-ID>)

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:

ALGOL NEWP

WAIT (<time>)

WAIT (<event-ID>)

Where <time> is the amount of time in seconds (or fractional seconds) that the task is to be suspended. In ALGOL the <time> must have its own set of parentheses.

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 may vary widely with respect to the time indicated in <time>. Smaller increments of time will have 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 some other task executes a CAUSE statement.

f. SIMPLE WAIT AND RESET FUNCTION:

ALGOL NEWP

WAITANDRESET (<event-ID>)

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:

NEWP

WAIT [DSABLE] (<event-ID>)

ALGOL

Not applicable

There exist some situations whereby certain operating system functions cannot be terminated while waiting on an event, such as waiting for an I/O complete or waiting for process synchronization. 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:

ALGOL NEWP

WAIT (<wait-parameter-list>)

WAITANDRESET (<wait-parameter-list>)

Where <wait-parameter-list> is a list of <event-ID>'s separated by commas. This list may also include a <time> parameter. The <time> parameter, if specified, must be the first parameter

The COMPLEX WAIT function allows a task to be suspended until any one event in the <event-list> is caused or until the <time> 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.

WAIT and WAITANDRESET are identical except for the state 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:

ALGOL NEWP

```
IF HAPPENED (<event-ID>) THEN <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. 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.

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. 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:

ALGOL NEWP

PROCURE (<event-ID>)

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:

ALGOL NEWP

FIX (<event-ID>)

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.

One should be careful in using the CONDITIONAL RESOURCE function to avoid the loop problem mentioned previously in the HAPPENED test.

c. LIBERATE FUNCTION:

ALGOL NEWP

LIBERATE (<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.

ALGOL NEWP

FREE (<event-ID>)

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:

ALGOL NEWP

IF AVAILABLE (<event-ID>)

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

The interrupt mechanism is not implemented in the NEWP language. Thus, the following discussion is limited to ALGOL.

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 statement. Associated with the interrupt mechanism are several interrupt functions:

a. INTERRUPT DECLARATION:

INTERRUPT <interrupt-ID>; <statement>

The INTERRUPT DECLARATION serves the purpose of naming particular interrupt code.

b. ATTACH FUNCTION:

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 compile-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:

DETACH <interrupt-ID>

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:

ENABLE <interrupt-ID>

DISABLE <interrupt-ID>

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, the initial state of an interrupt is ENABLED. 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:

```
ENABLED

DISABLE
```

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:

WAIT

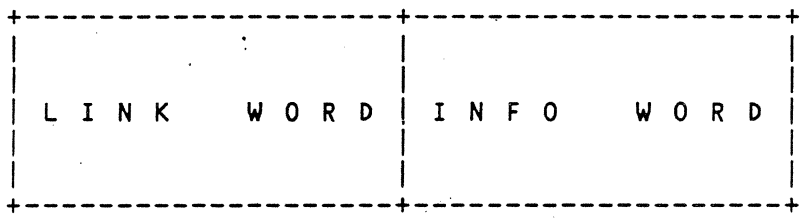
This function allows a task to be suspended until such time as any one of the task's interrupts is placed into execution. At completion of the interrupt code execution, control can only be returned via an explicit GO TO from the interrupt code to a label within the main program.

TIMETUNNEL

TIMETUNNEL is the MCP procedure called anytime a process wishes to wait so many seconds before continuing. 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. Starting at CALENDAR, GEORGE 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 9-6 is a TIMETUNNEL entry. This entry is part of TIMETUNNEL's portion of the process stack and may also be seen in figure 9-7. If this stack is a swapjob stack the entry will be outside of the stack in an area obtained by GETAREA.



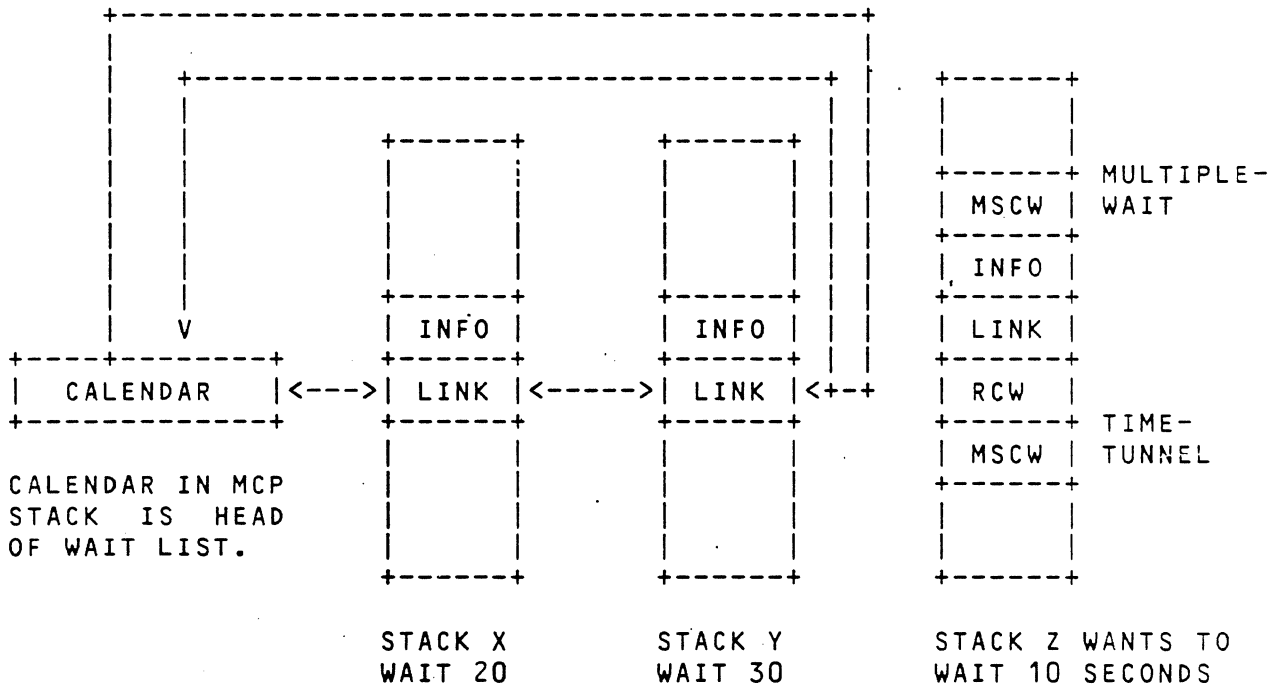
LINK WORD LAYOUT

```
[47:08] = CALTYPEF
      1 = CAUSESWAPOUT
      2 = CAUSEWAKEUP
[39:20] = QBACKF
[19:20] = QNEXTF
```

INFO WORD LAYOUT

```
[45:10] = CALSNRF
[35:32] = CALTIMEF
```

Figure 9-6. TIMETUNNEL ENTRY



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 9-7. TIMETUNNEL

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 9-7 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 9-8 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. This procedure, using CALENDAR, find the first entry and, seeing that the time has come, delinks the entry 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.

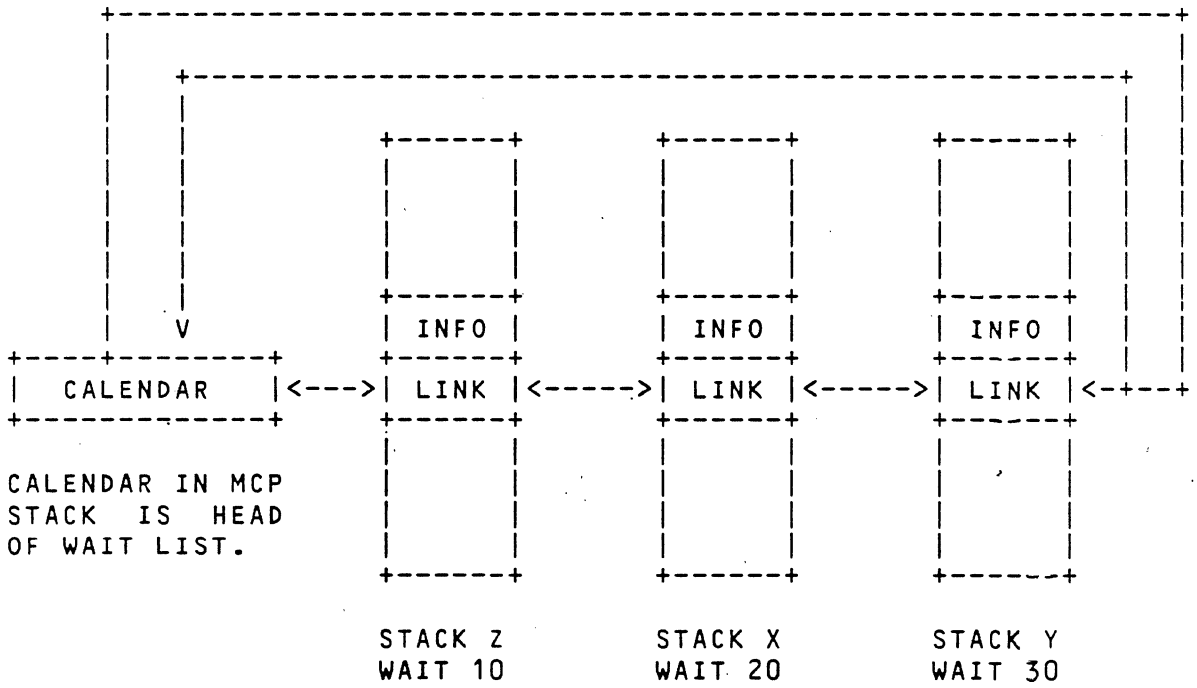


Figure 9-8. TIMETUNNEL

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 no-operation and an exit is made 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 move to another stack. This action can be seen in figures 9-9 and 9-10.

In figure 9-9 is an event in stack A. Stack B is waiting on this event. In figure 9-10, 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 two 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 9-10, 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.

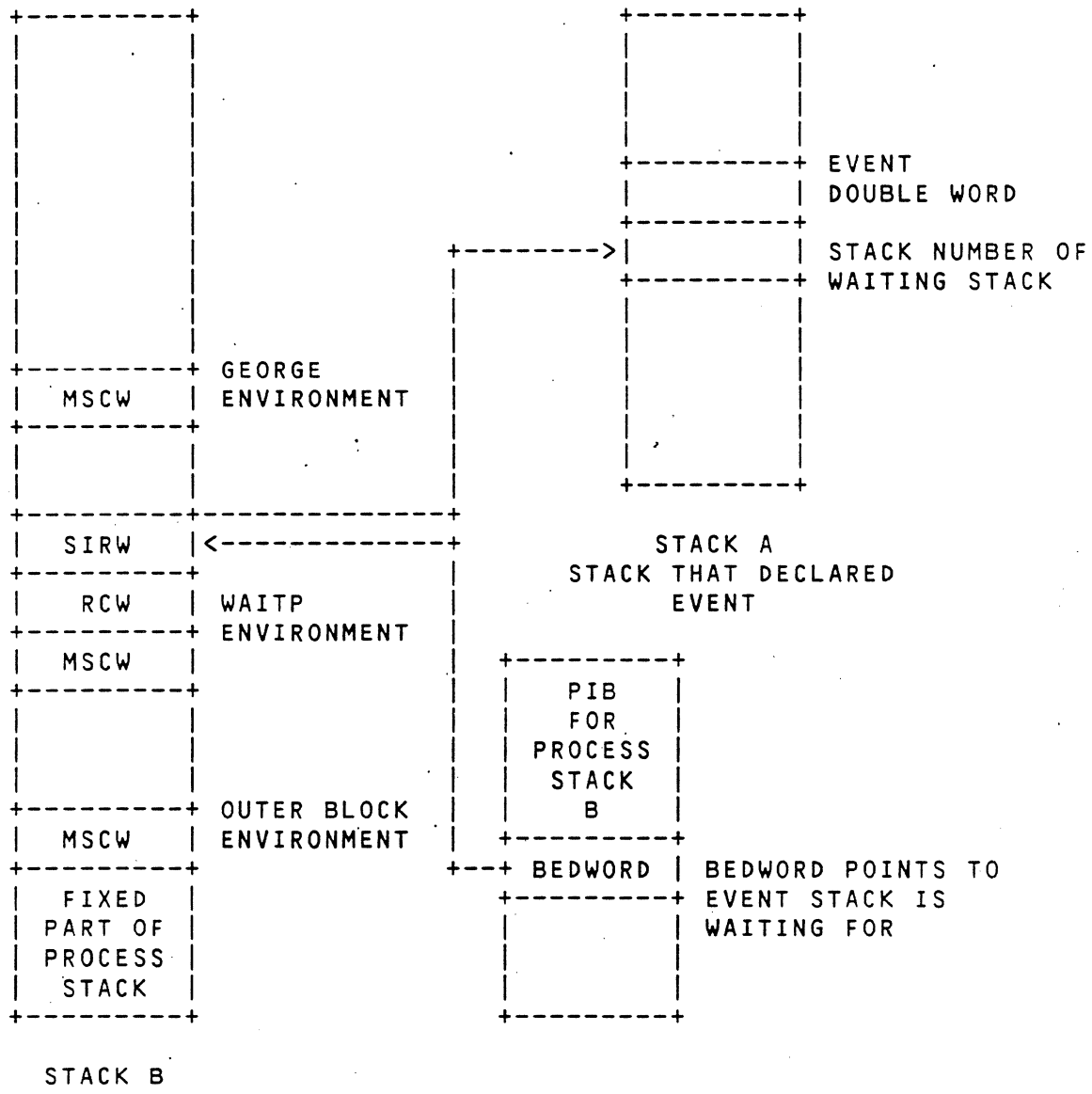


Figure 9-9. SINGLE STACK WAITING ON ONE EVENT

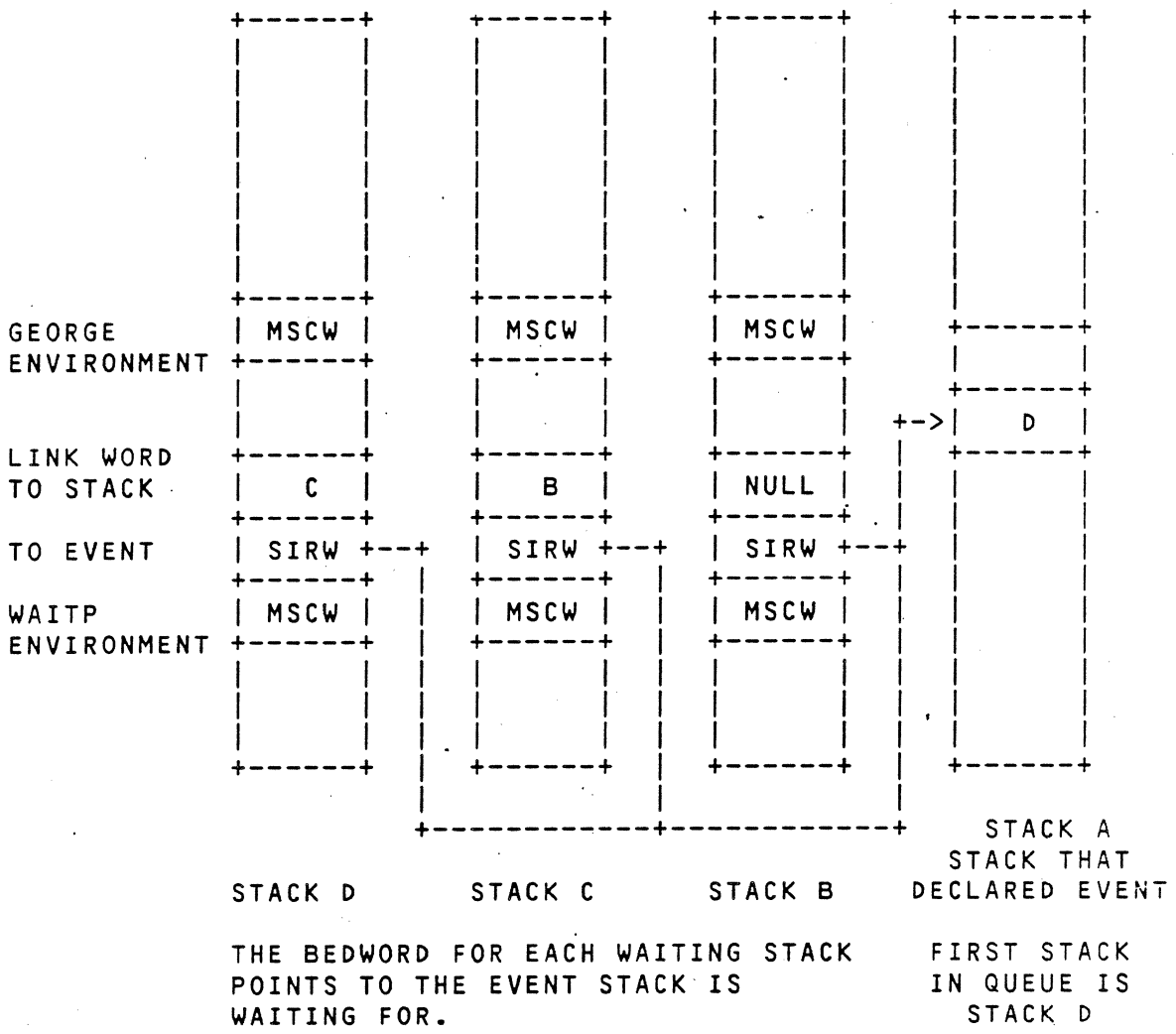


Figure 9-10. MULTIPLE STACKS WAITING ON SAME EVENT

PROCESS CONTROL EXAMPLE

This sub-section is an example of an ALGOL program initiating tasks. This program will be discussed followed by a brief discussion of the MCP procedures involved.

PROGRAM OPERATION

The program TASKINITIATION, shown in figure 9-11 and its associated external procedure WRITETOTAL, shown in figure 9-12, were included in this manual simply to provide an example program that initiates tasks and uses events. This program's operation will be discussed briefly.

```
BEGIN
   TASK
      TA,
      TB;
   EVENT
      CALLERUP,
      TABLELOCK;
   REAL
      CURROW,
      TABLESUM;
   DEFINE
      MAXINX = 09 #,
      ROWSZ  = 29 #;
   ARRAY
      TABLE[0:MAXINX,0:ROWSZ];

   PROCEDURE WRITETOTAL(TOT);
   VALUE TOT;
   REAL TOT;
   EXTERNAL;

   PROCEDURE INITTABLE;
   BEGIN
      REAL I,J;
      ARRAY REFERENCE ROW[0];
      FOR I:=0 STEP 1 UNTIL MAXINX DO
         BEGIN
         ROW:=TABLE[I,*];
         FOR J:=0 STEP 1 UNTIL ROWSZ DO
            ROW[J]:=I+J;
         END;
   END;
```

Figure 9-11. TASKINITIATION (1 of 2)

```
   REAL PROCEDURE GETROW;
   BEGIN
      PROCURE(TABLELOCK);
      GETROW:=CURROW;
      CURROW:=CURROW+1;
      LIBERATE(TABLELOCK);
   END;

   PROCEDURE ADDROWS(CALLED);
   VALUE CALLED;
   BOOLEAN CALLED;
   BEGIN
      REAL ROWINX,TOTAL,I;
      ARRAY REFERENCE ROW[0];
      IF CALLED THEN
         CAUSE(CALLERUP)
      ELSE
         WAIT(CALLERUP);
      WHILE ROWINX:=GETROW LEQ MAXINX DO
         BEGIN
         ROW:=TABLE[ROWINX,*];
         FOR I:=0 STEP 1 UNTIL ROWSZ DO
            TOTAL:=TOTAL+ROW[I];
         END;
      PROCURE(TABLELOCK);
      TABLESUM:=TABLESUM+TOTAL;
      LIBERATE(TABLELOCK);
   END;

   INITTABLE;
   REPLACE TA.NAME BY "PROCESSED.";
   PROCESS ADDROWS(FALSE) [TA];
   WHILE TA.STATUS NEQ VALUE(ACTIVE) DO
      WAITANDRESET(MYSELF.EXCEPTIONEVENT);
   REPLACE TB.NAME BY "CALLED.";
   CALL ADDROWS(TRUE) [TB];
   WHILE NOT (TA.STATUS EQL VALUE(TERMINATED) AND
              TB.STATUS EQL VALUE(TERMINATED)) DO
      WAITANDRESET(MYSELF.EXCEPTIONEVENT);
   TA.STATUS:=VALUE(NEVERUSED);
   REPLACE TA.NAME BY "OBJECT/WRITETOTAL.";
   RUN WRITETOTAL(TABLESUM) [TA];
END.
```

Figure 9-11. TASKINITIATION (2 of 2)

```
PROCEDURE WRITETOTAL(SUM);
VALUE SUM;
REAL SUM;
BEGIN
    FILE LINE(KIND=PRINTER);
    EBCDIC ARRAY REC[0:131];
   REPLACE REC BY " " FOR 132;
    REPLACE REC BY "TOTAL IS ",SUM FOR 5 DIGITS;
   WRITE(LINE,22,REC);
END.
```

Figure 9-12. WRITETOTAL, the External Procedure

The purpose of the program, TASKINITIATION, is to add all elements of a two dimensional array called TABLE and provide the total. In addition, the program will demonstrate task initiation.

The program, TASKINITIATION, begins executation by invoking the procedure INITTABLE. This procedure will place values in the two dimensional array called TABLE. Each element of the array is given the sum of its index values (TABLE[2,3] contains a 5).

Procedure ADDROWS is processed with a name of "PROCESSED" and a parameter of FALSE. ADDROWS is the procedure used to add the rows of TABLE. This procedure has one parameter (CALLED). If this parameter is TRUE the procedure causes an event (CALLERUP) otherwise it waits for this event to happen. This event is used to synchronize the copy of ADDROWS that was processed with the copy of ADDROWS that will be CALLed.

The outer block waits until the processed copy of ADDROWS is active.

ADDROWS is initiated with a name of "CALLED" and a parameter of TRUE. In this case a CALL is used to initiate ADDROWS. The outer block will wait for a CONTINUE (or EOT) from the CALLed copy of ADDROWS.

Because ADDROWS is initiated with a parameter of TRUE CALLERUP is caused. Both copies of ADDROWS are now active.

ADDROWS calls a procedure, GETROW, to return a row index into TABLE. The row index is obtained under control of a lock (event) called TABLELOCK. This lock will insure that a row index is used only one time and that all row index values are used.

ADDROWS will add all elements in the row. ADDROWS will continue this process until all rows are added.

ADDROWS adds the total to a global total, TABLESUM. This is done under control of TABLELOCK which will insure TABLESUM is updated by each total.

The outer block will receive a CONTINUE from the CALLed copy of ADDROWS. The outer block will wait until both tasks have terminated.

Procedure WRITETOTAL is RUN with a name of "OBJECT/WRITETOTAL" and the parameter TABLESUM. This procedure is EXTERNAL to TASKINITIATION and can be seen in figure 9-12. Because this procedure was RUN it is not dependent on TASKINITIATION. TASKINITIATION goes to EOT.

Procedure WRITETOTAL (figure 9-12) writes the parameter passed

(TOTAL) to a printer. WRITETOTAL goes to EOT.

The preceeding paragraph emphasized how TASKINITIATION 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

TASK INITIATION PROCEDURES

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 any parameters to be passed to the task followed by two undeclared parameters.

Shown below is a process statement extracted from the program TASKINITIATION (figure 9-11) and the machine language code. Figure 9-13 shows the stack for the program shortly after the call on DELIVERY.

```
PROCESS ADDROWS(FALSE) [TA];
   MKST
   NAMC (DELIVERY)
   ZERO              TYPE, 0 INDICATES A "PROCESS".
   NAMC (ADDROWS)    LIFE, PROCEDURE TO INITIATE.
   STFF
   ZERO              PASSED PARAMETER (FALSE).
   NAMC (TA)         TASK ARRAY (PIB).
   STFF
   NAMC (2,0)        BLOCK, SIRW TO CRITICAL
                     BLOCK MSCW.
   STFF
   ENTR
```

When DELIVERY was called, the parameter TYPE was passed with a value of 0 for PROCESS. If this had been a CALL then the TYPE 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 parameter was passed, and in this case, it was a ZERO (FALSE). After any procedure 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 9-14. shows how the MCP continues with the process request. The task (PIB) was declared in the outer block of TASKINITIATION and partially set up by INITIATEUSERTASK. As INITIATEUSERTASK continues, DOCTOR is called and then DOCTOR calls INITIATE which builds the process stack for ADDROWS and enters the stack number in the READY QUEUE or SCHEDULE QUEUE, depending on various resources.

A more detailed MCP procedure flow is show in figure 9-15. Notice that if ADDROWS had been called rather than processed the process stack of TASKINITIATION would have been topped off with GEORGE 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 TASKINITIATION.

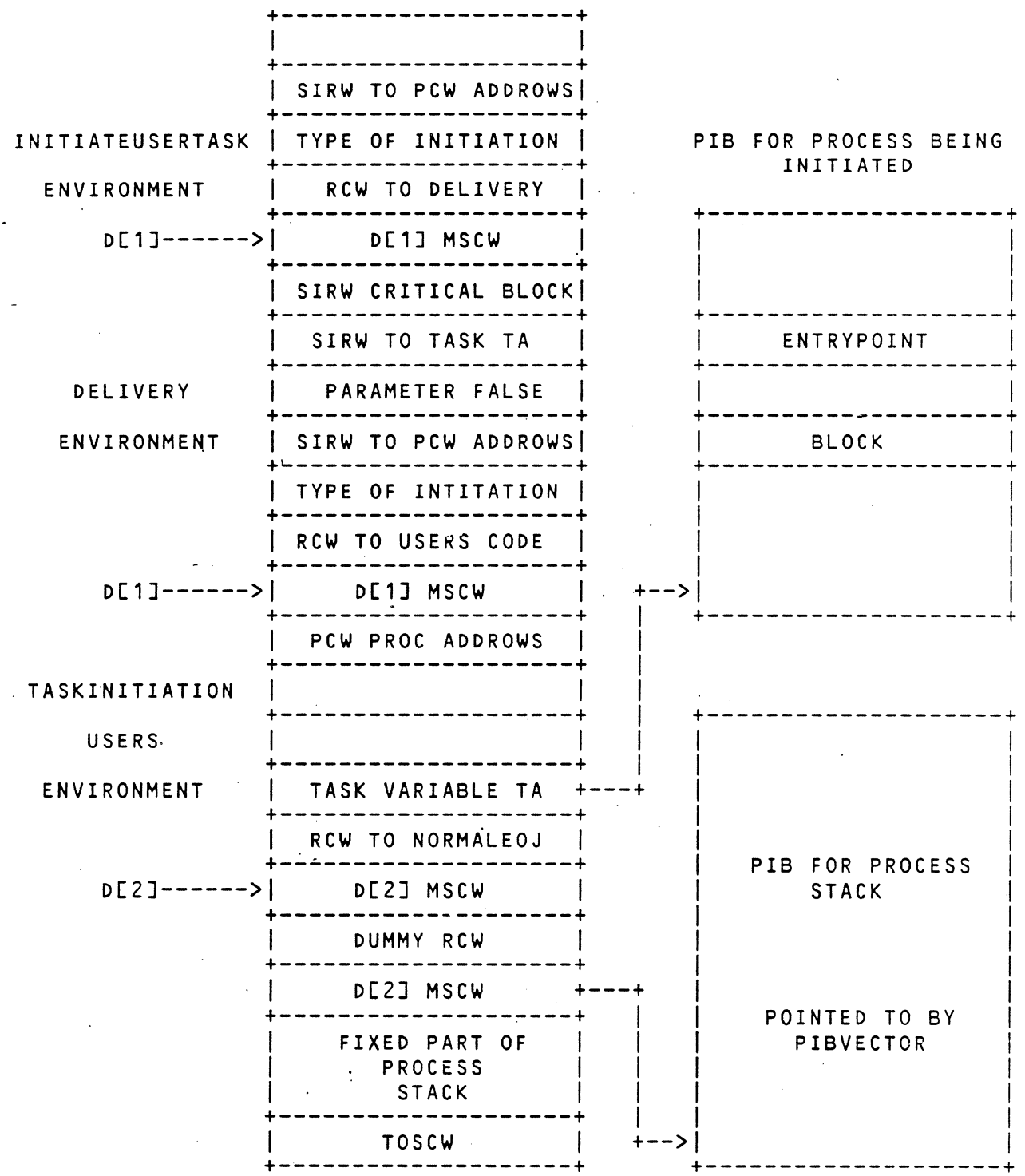


Figure 9-13. DELIVERY INITIATEUSERTASK Procedures.

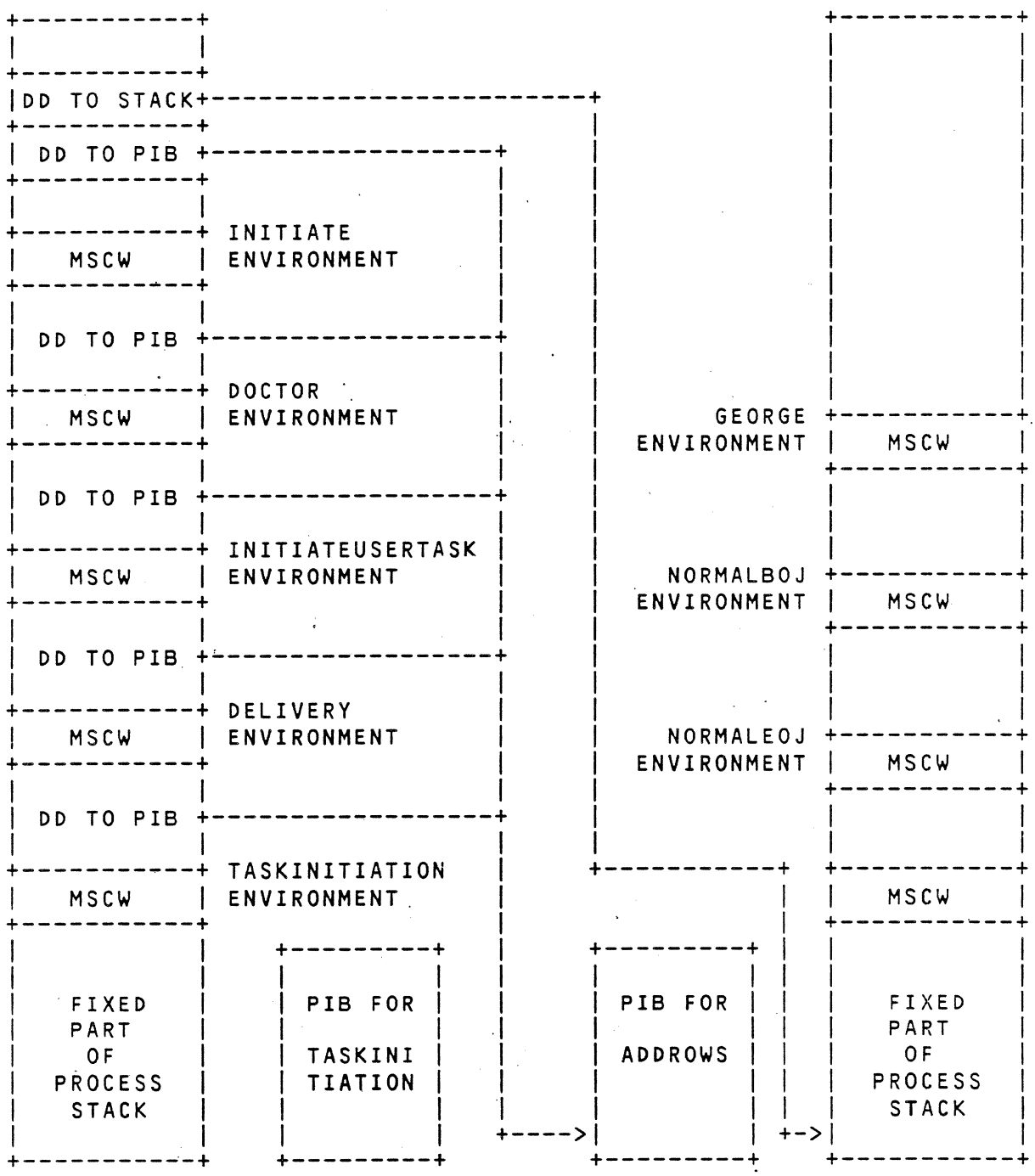


Figure 9-14. PROCESS STACK CONSTRUCTION

```
THE PROGRAM
DELIVERY
  INITIATEUSERTASK
    DOCTOR
      INITIATE
        PICKASTACK
        GETSPACE
        RESURRECT
          INSERTSTACK
    GEORGE (IF "CALL")
```

Figure 9-15. PROCEDURE FLOW FOR USER TASK INITIATION

OUTLINE OF DELIVERY

DELIVERY is an interface procedure which has two parameters. The procedure is designed to accept more than two parameters, it does not use the parameters above the two declared. DELIVERY calls INITIATEUSERTASK passing two parameters.

OUTLINE OF INITIATEUSERTASK

INITIATEUSERTASK will prepare the PIB for DOCTOR and compute the number of parameters based on MSCW linkage.

1. Chain down the LIFE parameter (SIRW) until a target is found. The target can be a PCW, operand or data descriptor.

   If a PCW is found this is an internal procedure (internal to this program) being initiated.

   If an operand is found, this is an external procedure being initiated. 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 initiate the library. After the library has been initiated the data descriptor will be replaced with an SIRW to a PCW (the user is initiating a procedure in a 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. PROCURE the lock PROCESSCHANGELOCK.

3. Get memory area for PIB if it has not been allocated.

4. Check to see if PIB is already in use. If the PIB is in use the initiator is DSED for INITIATE ACTIVE TASK.

5. Mark the PIB as in use by system.

6. If the process type is a RUN, get a new PIB memory area. The data from the old PIB is copied into the new PIB. The old PIB is marked as not in use by system.

7. LIBERATE the lock PROCESSCHANGELOCK.

8. HOSTNAME attribute is checked to determine if the program

is to run on a different host.

9. Move critical block MSCW (BLOCK) and SIRW to entry point PCW (ENTRYPOINT) to PIB.

10. Compute number of parameters based on MSCW linkage. In addition, INITIATEUSERTASK will look at the type of references these parameters make. This information is stored in the PIB. Some error checking is done.

11. The Software Control Word (SCW) above the critical block MSCW is found. The offset of the SCW is saved in the BLOCK word of the PIB.

12. Call DOCTOR passing the PIB as a parameter.

OUTLINE OF DOCTOR

The main task of DOCTOR is to assign task attributes to the PIB. In addition, DOCTOR will find the code file for an external process.

1. A boolean called EASY is set if the ENTRYPOINT SIRW points to a PCW. The user is initiating an internal process.

2. Check to see if a stack number is assigned to the PIB. If a stack number is assigned the initiator is DSED for INITIATE ACTIVE TASK. This check is made because not all tasks are initiated by calling INITIATEUSERTASK first.

3. If the task is to be initiated on another host a few checks are made to be sure the transfer is valid. These checks involve process type and parameters passed.

4. A check is made to be sure the user is not doing a run on an internal procedure (process type RUN, EASY and user). If this condition is detected the initiator is DS-ED for NON-EXTERNAL RUN.

5. Some words of the PIB are cleared of information left over from the last use of the PIB.

6. If the job attribute TASKLIMIT has been set, it is checked (maintained in the job's PIB).

   If the current TASKLIMIT is zero, the initiator is DSED for task limit exceeded. If the current TASKLIMIT is greater than zero, it is decremented by one and stored back in the job's PIB. If the type of initiation is RUN, the task being initiated will save the current value of TASKLIMIT and not start any more than this number of tasks.

7. 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 or set from a default value.

8. The ORGINALMTYPEF field of the SWAPSPEX word in the PIB is inherited. SWAPSPEX is the SUBSPACES attribute.

9. If the boolean EASY is set attributes are transferred from the parent. The attributes are USERCODE, NAME, COMPILERINFO (level of code file), sharing specification, STACKSIZE, code file header index, file parameter block (FPB), PRIORITY, OPTION, MAXPROCESSTIME, MAXIOTIME, PRINTLIMIT, PUNCHLIMIT, MAXWAIT, and DISKLIMIT.

10. If EASY is not set and the task is not to be initiated on another host a procedure named EXTERNALREFERENCE is called. EXTERNALREFERENCE will do the following:

    a. Build the name of the code file to be run. The MYNAME word in the PIB is used. If this word has not been set the binder information of the initiator is accessed to produce a name. If the binder information is missing the initiator is DSED for MISSING CODE FILE NAME.

    b. Procedure FINDAFILE is called to find the code file. One parameter tells FINDAFILE if it should wait for the file if it is not present (users wait but a MCS does not). FINDAFILE will return a -1 if the file was not found and the user is DSED for MISSING CODE FILE. If the file was found FINDAFILE will return the disk file header index of the code file header.

    c. If SUBSPACES was set to 2 for the initiation it is set to a 1 or 3. If the file is under a usercode it is set to a 3 otherwise it is set to a 1.

    d. The filekind of the file is checked to be sure it is executable. If the user is trying to compile, the code file is checked to be sure it is a compiler.

    e. Read in the Segment Zero of the code file.

    f. Be sure the code file is executable.

       The file could be code but not executable. That is, the file could be valid for binding only or not executable because it is unsafe NEWP code. Unsafe NEWP code can be initiated by the library linker (library called BYFUNCTION).

    g. Check any parameters. Both the number and type of each parameter is checked. If a mismatch is found

the initiator is DSED for PARAMETER MISMATCH.

h. If there is a PPB (Program Parameter Block) or FPB (File Parameter Block) 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.

i. The memory estimate is stored in the PIB. This information comes from a compiler estimate or code file average.

j. Any attributes in Segment Zero and not assigned to the PIB are assigned. These attributes include PRIORITY, MAXPROCESSTIME, MAXIOTIME, OPTION, TARGETTIME, STACKSIZE and RESTARTCOUNT. In addition, COMPILERINFO (mark level of code file), sharing specification, pointer to D1 stack image (D1STACKINFO) and entry point index are transferred from segment zero to the PIB.

k. If a job is being initiated, a restart area is built in the job file.

l. The tasks name is rebuilt to add the on pack part.

m. If this is a compiler being initiated, the compilee's name is set (YOURNAME). If no name can be found (file CODE not label equated) CODE is used.

11. If the type of initiation is RUN, the words for EXCEPTIONTASK, PARTNER, and BLOCK are cleared.

12. If the type of initiation is CALL and no partner is assigned, it is assigned to the initiator.

13. Some additional attributes are assigned if this is not an MCS or DBS. These attributes include MAXWAIT, ORGUNIT, backup destination, STATION, SUBSYSTEM, CHARGECODE, FAMILY, switches (set in WFL), ITINERARY and QUEUE assignment.

    If there is not a USERCODE assigned and the initiator has a USERCODE, the initiators USERCODE and ACCESSCODE is transferred.

    If there is an ACCESSCODE it is validated.

    If a CHARGECODE is required, USERDATA is called to validate the CHARGECODE.

    If the PIB does not have a BACKUPFAMILY assigned, it is assigned from the initiator. If the initiator does not have a BACKUPFAMILY, it is assigned from the DL specification.

14. The OPENCOUNT on the code file is incremented if the new task is to share the parents code file (internal process).

15. If the task is to run on another host, the PIB is placed in the external PIB vector. A task initiation message is built and sent to the other host using the BNA software. If the initiation was successful a BOT message is sent to the CONTROLLER and the EXTERNALFAMILYLINK is fixed up to include the new process. This will end the code for DOCTOR.

16. INITIATE is called passing the PIB.

17. 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

INITIATE will prepare a task for the READYQ. In general, INITIATE is called from DOCTOR, however independent runners are initiated with a call on INITIATE (DOCTOR is not involved). A task can go thru INITIATE more than once. The task could be scheduled, need to run in another box or be a swapjob. Each of these cases will cause a second trip thru INITIATE.

1. If this is the first pass thru INITIATE some set up must be done.

   a. Call PICKASTACK to assign a stack number (SNR). If this is not a job a mix number is also assigned. These numbers are placed in the SERIAL word of the PIB.

      A job does not need PICKASTACK to assign a mix number. The CONTROLLER will assign a mix number for a job.

   b. If the task to be initiated does not have a job file and needs one MAKEJOBFILE is called.

      An example of a task that does not have a job file would be a task initiated by ??RUN or MCS initiation by the DCC.

   c. If the task to be initiated has a job that ran thru a queue that had a priority limit and the priority of the task to be initiated is greater than the queue priority, the priority of the task to be initiated is set to the queues limit.

d. The stack number is entered in the JOBDATA array which is indexed by mix number.

e. The STACKINFO word is initialized with data that has been generated. This information includes task kind, visibility, box mask and other information. This array is indexed by stack number and protected by SILOCK.

f. The schedule priority is computed based on the type of process (normal user, IR and visibility) and our assigned priority. The priority is stored in the STACKSTATUS array which is indexed by stack number.

g. Some information about the type of references the new task can make are built. The information is based on parameters passed (are copy descriptors passed) and does this task access it's parents code file. Most of this information is generated by the compiler. This information is required for stack searching. The information is saved in the PROCESSFAMILYLINK word of the PIB.

h. If the new task has no parent the PIB is placed in PIBVECTOR as a MOM descriptor. The memory links are fixed up. Note that the initiator may or may not be the parent. The parent (if any) owns the critical block.

i. If the new task has a parent the following items are done.

   The PIB is placed in the PIBVECTOR as a COPY descriptor.

   The PROCESSFAMILYLINK word in the PIB is updated to contain our parent and siblings. The PROCESSFAMILYLINK word in our parents PIB is updated to contain our stack number as an offspring. The PROCESSFAMILYLINK is a linked list of related stacks.

   If this new task is dependent on its parent it is placed in the stack search graph for the parent (if the parent is searched so is this stack). In addition, the PROCESSCOUNTF of the Software Control Word (SCW) above the critical block MSCW is incremented.

j. Parameters in the calling sequence are moved from the stack to a memory area (GETAREA). The parameters will not be in the calling sequence if the task was initiated by WFL or ANABOLISM. The parameters are pointed to by TASKPARAMS.

k. Compute size of process (D2) stack from compiler estimate and other information. The process stack size is based on compiler estimate, number of parameters, fixed stack size and the size of the overflow area. Independent runners do not get the overflow area.

l. The estimated memory for this task is computed based on the compiler estimate divided by system factor 3.

m. The SUBSYSTEM and VISIBILITY attributes are used to determine which boxes the task can run in.

Box selection is based on run structure (library sharing specification, MCS, data base) and visibility requirements. The result of this code is to get a mask of boxes the task could run in. The actual box (if there is more than one) is selected later.

n. If a CM or HS is outstanding or OLAYSCOUT is waiting the stack will be scheduled.

The schedule code involves inserting the task in a queue of scheduled stacks. The linkage is made with the SCHEDDATA word of the PIB (also named READYON). The queue is protected by the lock SCHEDULELOCK. The MCS and CONTROLLER are told the task is scheduled. Later ETERNALIR will call INITIATE for the task again. The initiator is allowed to continue.

2. If the task is to be a swapjob it will be placed in SWAPPERS queue. INITIATE will exit back to the caller. SWAPPER will pick up the task initiation request in it's queue. After SWAPPER has set up a memory area in swapspace, SWAPPER will call INITIATE to continue the initiation process.

3. A BOT notice is sent to the CONTROLLER.

4. If a box has not been selected SCHEDULER is called to select the box from the box mask.

The primary consideration for box selection is memory requirements. If the task will not fit in any box it will be scheduled.

5. If the task is to be initiated in a different box than the one the initiator is running in, this processor can not continue with the initiation process (it can not see the memory for the stack). A message is sent to the ETERNALIR in the box where the task is to be initiated. ETERNALIR will get the task initiation message and call INITIATE from the proper box.

6. Call GETSPACE for process stack memory. If no memory is available, the task will be scheduled.

7. A stack vector entry is built and the stack is filled with hex BADBAD.

8. Move parameters from the area (GETAREA) to the stack.

9. The stacks priority is computed based on task type (IR [type of IR], MCS or normal task) and its base priority. The priority is placed in STACKSTATUS.

   Thus, a task priority is composed of more (type, locks procured, base and fine) information than can be set by a user. Priority stored in STACKSTATUS is the complement of the external priority (an internal priority of 0 is very high).

10. Build an IOCB in PIB (PIBIOCB) for use by MCP. If the system is tightly coupled or this is a swapjob GETAREA is called to get an area for EVENT waits. The area is pointed to by the BEDWORD in the PIB.

11. If this is a rerun of a checkpointed task the stack is set up to call CHECKPOINT after NORMALBOJ.

12. 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 GEORGE's environment (Figure 9-1).

13. Put PCW to GEORGE (PALACE) and SIRW to PCW (PALACEREF) in PIB.

14. Place a mask of which processors this stack can use in STACKSTATUS. This is based on which processors can see the stack.

15. Place the stack in the READYQ. From this point on the stack runs on its own. GEORGE will move to the stack and exit into NORMALBOJ.

16. INITIATE will exit.

OUTLINE OF NORMALBOJ

NORMALBOJ will finish task initiation (get a segment dictionary) and enter the users program. NORMALBOJ runs in an environment with the D1 register pointing to the PIB. This is done with a MSCW and PSEUDO stacks.

1. If a SUBSYSTEM was specified and the box the task is running in is not in the SUBSYSTEM a warning message is

produced (SUBSYSTEM was empty, undefined or inconsistant).

2. Set up an overlay file for the stack. If the stack is an independent runner it will use the MCP's overlay file (the MCP has an overlay file for each local box and global). If the task can see into it's parent stack it will share the overlay file with the parent. Otherwise the stack must set up an overlay file.

3. If this is an IR stack or the stack is DSED, a branch is made around the segment dictionary set up code. An IR uses the MCPSTACK as a segment dictionary and DSED stacks do not need a segment dictionary.

4. If the stack is an external process there may be a segment dictionary set up for this code file. The segment dictionary must be in the same box as the stack. If the process is internal it will use the parents segment dictionary. If there is a segment dictionary set up, this stack will add one to the RUNNINGCOUNT and link to it.

5. If there is not a segment dictionary this stack can use, it must be set up as follows:

   a. Call PICKASTACK to get a stack number.

   b. Initialize the STACKINFO entry.

   c. Get a memory area for the PIB for the segment dictionary. The PIB for a segment dictionary is shorter than a PIB for a process stack. The PIB is cleared and a few words are set up.

   d. GETSPACE is called to get memory for the segment dictionary. If memory is not available a message for words required is produced. The operator may enter OK or DS.

   e. The segment dictionary is entered into the stack vector. The base of the stack is initialized.

   f. Read the segment dictionary into memory using a disk read with tags operation.

   g. Change any MCP intrinsic references to SIRW's that point into the MCP stack. User programs do not contain direct references to MCP procedures.

   h. The segment dictionary is linked into the list of segment dictionaries that represent the same code file.

   i. Set the RUNNINGCOUNT of the segment dictionary

(stored in the PIB) to one.

6. The segment dictionary stack is added to the stack search graph for the process stack. The segment dictionary stack is also added to the programdump graph for the process stack.

7. If this stack has a job stack with limits (MAXIOTIME, PUNCHLIMIT, DISKLIMIT ect.) they are moved from the job to the PIB.

8. The TASKFILE is set up for the stack.

9. An SIRW to the entry point PCW is created and placed in the PIB at ENTRYPOINT. The entry point PCW is changed to a RCW with proper stack number and placed in the stack. The RCW becomes NORMALBOJ's RCW.

10. If the stack is visible the EXCEPTIONEVENT is caused, a BOJ message is sent to the CONTROLLER, a BOT record is written to the log and a BOJ message is sent to the MCS.

11. If the code file is more than 3 releases old it is DSED. If the code file is 2 releases old a warning is given (this code file will not run on the next release).

    If the program is XALGOL it is DSED.

12. Exit into the users program using the entry point RCW.

13. The stack will go thru stack building code and start running.

OUTLINE OF NORMALEOJ

NORMALEOJ will prepare a stack and PIB for termination. By the time NORMALEOJ is entered the users part of the stack is cleared.

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 was linked to a DBS, records owned by this stack must be released. Each DBS this stack was linked to 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 visible 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 and STACKSIZE for the code file are updated. The memory estimate update is used for scheduling the task the next time it is run. The STACKSIZE update is used the next time a process stack is built for this code file (avoids stack stretch).

11. The RUNNINGCOUNT of the segment dictionary is decremented. If it is zero, the segment dictionary is terminated by calling TERMINATED1STACK. The segment

dictionary stack is removed from graphs for this stack.

12. If the overlay file being used by this stack is owned by this stack it is released (the disk is returned).

13. The owner of the stack is changed to the MCP.

14. FORGETCHECK is called to make sure there are no in-use areas of memory left around. A search of memory is made if the COREINUSE is not zero. If any memory areas are found they are marked as save and a memory dump may be taken.

15. Mix numbers are cleared from the PIB.

16. A EOJ message is sent to CONTROLLER. In the case of job stack, the CONTROLLER will start the printing.

17. If this stack has a job stack with limits (PUNCHLIMIT, PRINTLIMIT, DISKLIMIT ect.), the limits are changed to allow for what this task used.

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. SWAPPER will call TERMINATE.

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

TERMINATE will remove the stack and finish with the PIB. TERMINATE is called by ETERNALIR or SWAPPER.

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 as follows.

   a. If there is a critical block for this stack, the PROCESSCOUNT in the SCW above the critical block MSCW will be decremented by one.

   b. The stack is removed from any graphs it is in.

   c. The stack is removed from the PROCESSFAMILYLINK.

d. The EXCEPTIONEVENT is caused for this stack. It is changing status.

e. If the type of process was a CALL, a CONTINUE is done to the CO-ROUTINE.

f. If the parent is suspended an OK is given.

3. If the stack was not a SWAPJOB, the memory area for the stack is released.

4. The entries for STACKINFO, STACKSTATUS, PIBVECTOR and STACKVECTOR are cleared. The stack number is returned.

5. An event, STACKFINISHED, is caused.

OUTLINE OF TERMINATED1STACK

TERMINATED1STACK will remove a segment dictionary stack.

1. The code file header is released.

2. If the segment dictionary is linked to the intrinsics it is delinked. If this is the last stack using the intrinsics the stack will be terminated by calling TERMINATED1STACK.

3. Search the stack for any present descriptors. The memory area for a present descriptor is released.

4. Remove the segment dictionary from the stacksearch graph.

5. Call FORGETCHECK for the stack.

6. Release the memory area for the PIB.

7. If the segment dictionary is not swapable call FORGETSPACE to release the memory area.

8. Clear STACKINFO, STACKSTATUS, PIBVECTOR, and STACKVECTOR. The stack number is returned.

OUTLINE OF WAITP

WAITP is called when a process wants to wait for an event to happen with the syntax WAIT(EV). The compiler will pass WAITP two parameters. The first is an SIRW or indexed data descriptor pointing to the event. The second is information for WAITP (happened state, is task DS-able and is task swapable while waiting).

1. If the event has happened and the user did not want the event reset, WAITP will exit back to the caller. This statement has the side effect of touching the event.

2. Build the parameter for GEORGE from information in second parameter. GEORGE is called later.

3. The READYQ is locked (BUZZED).

4. If the event has happened WAITP and the caller wants the event reset, it is reset. The READYQ is unlocked and WAITP exits.

5. If the stack is DS-able and the stack is DSED WAITP will exit.

   BNA has the ability to be DSED and run in a normal way. BNA sets a bit in the HISTORY word which allows the DS to be ignored.

6. If this is a user calling WAITP a wait clock is started. This is used to time stacks waiting on events that have specified a WAITLIMIT.

7. The status is set to waiting.

8. The stack number of the first stack waiting on this event is placed in this stack (event queue linkage is second parameter). The stack number comes from the event word.

9. The stack number of this stack is placed in the event word. Thus, a linked list of stacks is formed.

10. If this task is a swapjob or the system is tightly coupled, a reference to the event and the event linkage is stored in the area (GETAREA) pointed to by the BEDWORD. Otherwise, the BEDWORD contains the address of the event in the stack.

11. The stack offset to the event MSCW is stored in the BEDWORD.

12. WAITP calls GEORGE passing the parameter. GEORGE will

process switch to another stack. When the event has happened GEORGE will exit back to WAITP which will exit back to the caller.

OUTLINE OF PROCUREP

PROCUREP is called when a user wants to procure (get control of) an event. It is called by the syntax PROCURE(EV). The compiler will pass one parameter, an SIRW or indexed data descriptor to the event.

1. The event is touched to be sure it is present.

2. The READYQ is locked.

3. If the caller is the MCP the priority of the stack is increased.

4. If the event is not available the following is done.

   a. The stack state is set to waiting.

   b. If this is a swapjob or the system is tightly coupled a reference to the event is stored in the area (GETAREA) pointed to by the BEDWORD. The link word is also stored in the area.

   c. If the caller is a user a wait clock is started for WAITLIMIT time check.

   d. The stack is linked into the queue of stacks waiting for or wanting to procure the event. The stack is linked at the end of all stacks waiting and in priority order of all stacks wanting to procure the event.

   e. The event linkage (stack number and procure information) is stored in the stack or area pointed to by the BEDWORD.

   f. GEORGE is called to allow the processor to switch to another stack.

   g. When GEORGE exits back to this stack it (the stack) owns the lock. PROCUREP will exit back to the caller.

5. If the event is available it is marked as unavailable. The owner stack number (SNR) is placed in the event word. The READYQ is unlocked and PROCUREP exits.

OUTLINE OF LIBERATEP

LIBERATEP is called when a user wants to release a lock. The syntax LIBERATE(EV) will call LIBERATEP passing an SIRW or indexed data descriptor to the event.

1. The event is touched to be sure it is present.

2. The READYQ is locked.

3. The first word of the event is rebuilt with the happened bit on and the event marked as available.

4. If the caller is the MCP the priority of the stack is decreased (it does not own the lock now).

5. If there are no stacks in the event linkage and the second word of the event is zero, the READYQ is unlocked and an exit is done to the caller. The second word of an event is used for linkage to a software interrupt.

6. If there are stacks in the event linkage or software interrupts attached, CAUSEP is called passing the event and the fact that the READYQ is locked.

OUTLINE OF CAUSEP

CAUSEP is called by the user to make an event happen and wake up stacks waiting on the event. The syntax CAUSE(EV) will call CAUSEP passing an SIRW or indexed data descriptor to the event. A second parameter provides the happened state of the event. This parameter is also used to indicate the READYQ is locked. CAUSEP can be directly called by LIBERATEP to unwind the event queue linkage.

1. If the caller is not LIBERATEP the READYQ is locked. The event happened state is set as per the parameter.

2. Loop over all stacks in the event queue linkage.

   a. If the stack said to reset the event it is reset.

   b. If the event is unavailable and this stack wants to procure the event, the stack is left at the head of the queue (only a CAUSE was done). This is the end of the loop.

   c. If the event is available and this stack wants to procure the event, this stack is marked as the owner of the event and the event is marked as unavailable. The stack is placed in the READYQ by calling RESURRECT.

d. If this stack is waiting for the event it will be placed in the READYQ by calling RESURRECT. The wait clock is zero'ed.

3. All stacks in the event linkage are processed. If there are software interrupts attached to the event they are processed.

4. When the last stack is placed in the READYQ by calling RESURRECT, RESURRECT is told to be sure the highest priority stack gets the processor.

5. The READYQ is unlocked.

OUTLINE OF RESURRECT

RESURRECT is called to insert a stack into the READYQ. The READYQ linkage starts at READYQHEAD and is made thru the STACKSTATUS array. For each entry in the STACKSTATUS array the upper bits are priority and the lower bits are the READYQ linkage. The stack is inserted into the READYQ in priority order.

1. The fine priority of the stack is adjusted.

2. If there is a processor in the stack, the stack state is set to ALIVE. If the stack being RESURRECTed is not this stack, the processor is interrupted. This will cause the stack to run thru GEORGE. GEORGE will see the stack is ALIVE and allow it to run.

3. If the stack is in memory it is inserted in the READYQ in priority order. If the stack is on disk, SWAPPER is told to swap the stack in.

4. If RESURRECT is told to be sure the highest priority stack gets the processor, it may interrupt other processors.

OUTLINE OF GEORGE

GEORGE is the procedure that gives up the processor to another stack (GEORGE does the process switch). GEORGE is also the procedure that receives the processor back.

GEORGE is a define that calls a procedure called GEORGE declared in SOPHIA. The call to GEORGE is made VIA the word at PIB[SNR,PALACEREF]. One parameter is passed to GEORGE. The syntax will cause the following code to be generated. MKST, NAMC (PIB), RPRR (SNR), NXLN, LT16 (PALACEREF), INDX, LODT, VALC (parameter), ENTR.

The ENTR will be done on the word in the PIB at PALACEREF. This word is an SIRW to the PALACE word in the PIB. The PALACE word is a PCW to GEORGE. The SIRW and PCW that are used have stack numbers (PSEUDO stack numbers) where the PIB resides in memory. When the entry into GEORGE is made the D1 register will point to the base of the PIB. This will allow GEORGE to access the PIB entries without index operations. GEORGE is called with the READYQ locked.

Steps 1 thru 3 are done if this is not an IDLER (not ARP [Assocication of Retired Processors]).

1. The PROCESSTIME the task has picked up in this slice is computed. This is added to the total PROCESSTIME.

2. If the stack has a PROCESSTIME limit, it will be checked. If it is too large, the stack will be DS-ED.

3. The fine priority is adjusted for the task. This will give a better priority to tasks which do not use much processor time. The actual calculations are documented in the MCP.

4. If it has been greater than one second since ONESECONDBURDEN has been called, it is called. The procedure is outlined below.

5. Check to see if time is up for stacks waiting on time. If the time is up, wake up the stacks.

6. If the stack is not alive and it is a swapjob consider swapping it out.

   The stack may not be able to be swapped. The parameter to GEORGE may hold it or there could be I/O operations or offspring outstanding.

7. If the stack state of this stack is ALIVE, it is placed in the READYQ in priority order. If the stack WAITING (not ALIVE) it is not placed in the READYQ.

8. The processor will 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 and not blocked by stacksearch. Not all processors can run all programs (an AP can not run a user task). A stack will always be found. If a user does not want the processor the IDLER will run.

9. If the stack found is not this stack, a MOVESTACK is done. The MOVESTACK will change addressing environment (on a B7900 it also changes CENR and DENR). The code registers will not change. GEORGE will continue to execute.

10. The stack is clocked on the processor.

11. The ACTIONQ for the stack is checked. The ACTIONQ may tell GEORGE to delink events, call KANGAROO or process software interrupts.

12. The READYQ is unlocked.

13. The interval timer is set for about one second.

14. Exit to user.

OUTLINE OF IDLER

The IDLER stacks will run when no other stack wants the processor. IDLERS run at a very low priority (last stack in READYQ). The IDLERS are set up at initialization and run until a processor is removed from the system. The IDLER is an infinite loop and shares a close relationship with GEORGE. The IDLER is really a procedure called PAWS in SOPHIA.

1. The interval timer is set.

2. IOM's in the same box as this CPM are told to interrupt this CPM on each I/O operation.

3. The idle pattern is displayed and the processor is placed in a SUSPEND state (hold until interrupt). At some point the processor will get an interrupt and continue processing.

4. Enter normalstate to allow interrupts. This will cause the processor to enter HARDWAREINTERRUPT. The interrupt will be processed and GEORGE will be called. GEORGE may process switch or exit back to the IDLER loop.

5. Continue to run the loop.

OUTLINE OF ONESECONDBURDEN

ONSECONDBURDEN runs once a second. It will do some of the periodic functions of the MCP.

1. Reset the EGG timer.

2. Cause STATUSCHANGEEVENT every 2 to 10 seconds.

3. Check for hung processors.

4. Check for peripheral status change. If a status change has occured a process may be forked to handle it. For

example, a tape goes ready (READALABEL).

5. Move memory dump from disk to tape if required.

6. Write the memory log to disk after a halt load.

7. Update utilization statistics.

8. Check for hung I/O's.

9. If the time of day is greater than one day, change the date.

OUTLINE OF KANGAROO

KANGAROO is called to get control or interrupt the normal flow of a task. For example, it will be called if a task is DSED.

KANGAROO is passed three parameters. WHO is the stack which is to receive the the action, CRIME is what the action is (DS, ST, DP, QT ...). REASON is a HISTORY word.

A stack can go thru KANGAROO more than once. That is, the operator could ST a program and then DS the program. This would cause two calls on KANGAROO.

1. If the stack to receive the action is not this stack (a KANGAROO is not being done on this stack) the following is done.

    a. Call GETAREA to get a two word area. In this area store the KANGAROO parameters (CRIME, REASON and call KANGAROO).

    b. The area (GETAREA) is linked into the ACTIONQ of the PIB that is to receive the KANGAROO. A stack will look at its ACTIONQ when it goes thru GEORGE. It will see the request to call KANGAROO. It calls KANGAROO using the parameters from the area. KANGAROO is building a summons.

    c. If the stack is to be DSED the following is done to move the process along.

        If the stack is scheduled it is marked as FS'ed (XS'ed) and ETERNALIR is sent a message telling him to initiate the stack.

        If the stack is selected (being initiated by ETERNALIR), the priority is increased.

        If the stack is waiting, the priority will be increased. If the stack is a library it is resumed. Otherwise, the stack is RESURRECT'ed. This will cause a trip thru GEORGE who will look at the ACTIONQ. The stack may or may not really wake up depending on what event it is waiting on (user or MCP event).

        If the stack is ALIVE, the processor that is running the stack is interrupted. This will cause a trip thru GEORGE.

        If the stack is in the READYQ, the priority will be increased and it will be relinked in the READYQ.

d. If the stack is to be ST'ed a bit is set in STACKINFO which indicates the stack has been requested to stop.

2. The following code is executed when the stack that is to receive the KANGAROO is the stack running KANGAROO. This may be a direct call or a call from GEORGE looking at the ACTIONQ.

3. The HISTORY word is placed in the PIB.

4. The reason (CRIME) for the KANGAROO call is added to the INTERSESSION word of the PIB.

5. A loop is started to look down the stack for a point to INTERLOOP. KANGAROO will not INTERLOOP a RCW which points to MCP code or one in a segment with the DONTDS (DMSII accessroutines) bit on. In addition, KANGAROO will not INTERLOOP a BNA block. Only entered blocks (MSCW's) can be INTERLOOPED.

6. The RCW found is removed from the stack and placed in the PIB at INTERCEPTEDRCW. A RCW to INTERLOOPER (with the correct lex level) is placed in the stack where the users RCW was.

7. If the stack is to be DSED the following is done.

   a. If MCPTEST is set and TERMINATE is reset a memory dump is taken.

   b. All other CRIMEs are cleared from the INTERSESSION word in the PIB. DS has the highest priority.

   c. The STOPPOINT (where the user lost control, INTERCEPTEDRCW) is stored in the PIB.

   d. If the stack is waiting on its reply event, the reply is set to DS and the REPLYEVENT is caused.

   e. If the stack is not going to ignore the DS (not BNA), it will increase its priority and try to delink interrupts. Stacks waiting on some MCP events may not be able to be delinked.

      A BNA stack can ignore a DS and continue to run in a normal way.

8. This ends KANGAROO at some point the stack will begin to exit from the procedures that were called. The stack will exit into what was once user code. However, the user RCW has been replaced with a RCW to INTERLOOPER. The stack will begin executing INTERLOOPER.

OUTLINE OF INTERLOOPER

INTERLOOPER is an interface routine. It calls INTERCEDE with a parameter of TRUE. This routine must not declare any local variables (it runs at an unknown lex level).

OUTLINE OF INTERCEDE

INTERCEDE is where the action of the KANGAROO call takes place. The MCP now has contol of the stack.

1. The users RCW is replaced in the stack.

2. A CASE statement is executed based on the FIRSTONE of the INTERSESSION word. The CASE statement will select one of the following.

   a. TOSTRETCH will try to do a stack stretch (the stack had a stack overflow).

   b. TODUMP will call PROGRAMDUMP.

   c. TOSWAP will request a swap out (swapjob).

   d. TOGROW will increase subspace size.

   e. TOSLICE will time slice the stack.

   f. TOST will stop the stack. The stack will cause its EXCEPTIONEVENT and set up a reply of OK or DS. A message is produced (OPERATOR STOPPED ect.). The stack waits on its REPLYEVENT. When the OK or DS is entered the REPLYEVENT is caused.

   g. TODS will DS the stack. PROCESSKILL is called to process the stack. PROCESSKILL does not return.

   h. TOSOFTINT will process software interrupts.

   i. TOBR will take a operator requested checkpoint.

3. INTERCEDE will exit back to the users code. However, if PROCESSKILL was called it will not exit (back to INTERCEDE).

OUTLINE OF PROCESSKILL

PROCESSKILL will prepare a stack to enter NORMALEOJ. It may be called from INTERCEDE or directly from any MCP routine.

1. A boolean called ERRTERM is set to not no fault of his. That is, ERRTERM is true if the user faulted.

2. If the stack owns PROCESSCHANGELOCK it is liberated (a stack should not die with a lock).

3. If a STOPPOINT has not been set up in the PIB, it is set from the INTERCEPTEDRCW if there is one. If there is no INTERCEPTEDRCW, PROCESSKILL searches down thru the stack for a user RCW.

4. A memory dump may be generated for MCP FAULT LOCKED (fault in MCP code with a lock locked), FAULT IN DO CODE (fault in MCP code) or TERMINATE RESET (the stack faulted and TERMINATE was reset).

5. If the stack faulted and the task option fault is set or the stack was DSED and the task option DSED is set a programdump is taken.

6. The stack history is generated (RCW and LINEINFO trace).

   Before the stack history is generated a bit (NOHISTORY) is checked. This bit will tell PROCESSKILL if there has been an attempt to generate the history. If the attempt has been made PROCESSKILL will not try to generate the history again. This code will prevent a recursive fault condition. That is, we fault while generating the history and do not want to do it again. A stack overflow in the MCP could result.

   If there is not enough room in the users stack, ETERNALIR will be told to get the stack history information.

7. If this stack is a DBS, all users of the data base are DSED.

8. If this stack has offspring, they are DSED for DEATH IN THE FAMILY.

9. If there are external offspring (running on another host), they are DSED by calling XENOCIDE.

10. Wait for all offspring (that were DSED) to go away.

11. If this stack is a library being DSED, it will DS all users of the library and wait for them to go away.

12. If the stack faulted and has a non zero RESTARTCOUNT, AUTORESTART is called to restart the program.

13. If the stack does not have a RESTARTCOUNT, GOTOSOLVER is called. GOTOSOLVER will do a BLOCKEXIT on each block and exit into NORMALEOJ.

SECTION 10

INPUT/OUTPUT OPERATIONS

INTRODUCTION

This section is divided into 2 sub-sections. The first sub-section is an overview of the I/O subsystem, the second is concerned with a program named FILEEXAMPLE and provides an account of I/O procedure flow. This sub-section is followed by several procedure outlines.

I/O SUBSYSTEM OVERVIEW

There are two main characteristics of the logical I/O design which are responsible for its speed and obscurity. These two characteristics will be discussed in this sub-section.

The memory area containing the File Information Block (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 (D1) will be pointing at the base of the FIB. Thus, they can address FIB variables directly as stack locations (1,X) rather than FIB[X] where "FIB" is an array. This means that NEWP can do VALC(1,X) instead of NAMC FIB, LT8 X, NXLV. The VALC is clearly more efficient.

The other 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 the 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 and space.

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. 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 for 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. There is also a read-serial column routine and read-random column routine. 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 and translation. These decisions select one line of code rather than another, or one path rather than another. Individually these decisions 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.

FIB USED AS A STACK

The other 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 which 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".

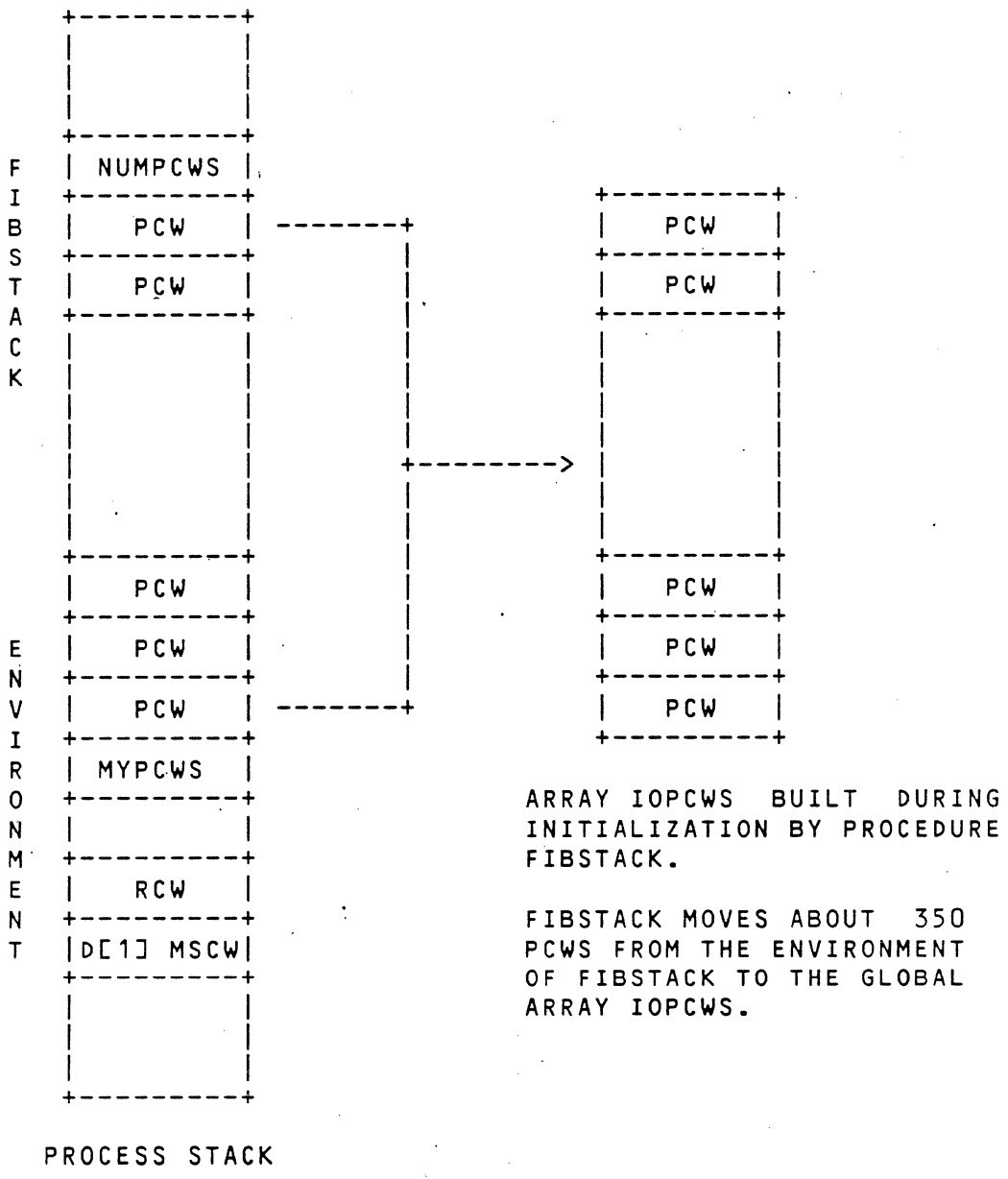


Figure 10-1. IOPCWS Array

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 which 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 point to the MSCW in the FIB. The D2 register, of course, addresses the MSCW in the process stack (placed there by the MKST 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.

FILE EXAMPLE PROGRAM

The program, FILEEXAMPLE, shown in figure 10-2 will be used as an example of how the MCP handles user I/O requests. The program does little more than writing and reading a disk file but in doing so, it opens up many areas of discussion.

FILEEXAMPLE declares a file named FYLE, an array named EA and an integer named I. The program does not do an explicit OPEN on the disk file. File FYLE is opened with an implied OPEN on the first write to the file. The program builds five records and writes them to the file. A random read of record one is performed and a PROGRAMDUMP is taken.

```
$ SET LIST STACK CODE
BEGIN
   FILE FYLE(KIND=DISK,NEWFILE,AREASIZE=100,
    MAXRECSIZE=30,BLOCKSIZE=60,TITLE="DISK/FILE.");
   EBCDIC ARRAY EA[0:179];
   INTEGER I;
   REPLACE EA BY "DATA RECORD NUMBER:"," " FOR 161;
   FOR I:=0 STEP 1 UNTIL 4 DO
      BEGIN
      REPLACE EA[20] BY I FOR 1 DIGITS;
      WRITE(FYLE,30,EA);
      END;
   READ(FYLE[1],30,EA);
   PROGRAMDUMP(ARRAYS,FILES);
END.
```

Figure 10-2. FILEEXAMPLE

COMPILER DESCRIPTION OF THE FILE

When the program is compiled information in the file declaration is placed in a data pool and written to the program's code file. This data pool is an encoded form of the file declaration and is shown in figure 10-3.

The first word of the data pool contains the software level of the file and the language the file was declared in. The second word contains the internal name (INTNAME) of the file in standard form. In this example the INTNAME is 8 characters long, the security byte is a 01 (user file), there is one identifier in the name, the name is four characters long and the name is FYLE. The internal name is followed by an attribute list.

The attribute list contains entries in two different formats. A numeric or boolean attribute is represented by a variable length entry of the following format: length byte, attribute number, attribute value. For example, the hex string 4'030801' means: the entry is 3 characters long, the attribute number is 8 (KIND) and the value is 1 (DISK). A string attribute is represented by a variable length entry of the following format: 02, attribute number, length of string, string. In each of these formats the first two (numeric or boolean) or three (string) entries contain a fixed number of characters and the last entry contains the variable length part. Examples of both formats can be found in figure 10-3.

When FILEEXAMPLE is executed, its stack building code will generate a data descriptor that points to this data pool. This data descriptor is placed in the process stack at the address couple assigned to the file. As far as a process stack is concerned, a file is simply a data descriptor and nothing else.

WORD	CONTENTS	COMMENTS
0000	010000000000	SOFTWARE LEVEL AND LANGUAGE
0001	08010104C6E8	INTERNAL NAME OF FILE IN
0002	D3C500000000	STANDARD FORM
0003	031D04030801	ATTRIBUTE LIST
0004	038801031164	
0005	030F1E030E3C	
0006	02000AC4C9E2	
0007	D261C6C9D3C5	
0008	4B0000000000	

In this example the attribute list contains:

```
03  There are three characters in this entry.
1D  INTMODE, a default attribute.
04  Value of INTMODE is 4 (EBCDIC).

03  There are three characters in this entry.
08  KIND attribute.
01  Value of KIND is 1 (DISK).

03  There are three characters in this entry.
88  NEWFILE attribute.
01  Value of TRUE.

03  There are three characters in this entry.
11  AREASIZE attribute.
64  Value of AREASIZE attribute in hex.

03  There are three characters in this entry.
0F  MAXRECSIZE attribute.
1E  Value of MAXRECSIZE in hex.

03  There are three characters in this entry.
0E  BLOCKSIZE attribute.
3C  Value of BLOCKSIZE attribute in hex.

02  This is a string attribute.
00  TITLE attribute.
0A  The string is 0A (10) characters long.
    The string is "DISK/FILE.".
```

Figure 10-3. Data Pool For File FYLE

OPENING THE FILE

The data descriptor for file FYLE (built by stack building code) is set up with an address which points into the code file and a length which represents the length of the data pool. The size field of the descriptor contains a seven, which is not valid. The index bit of the descriptor is also set. This in itself is not incorrect but an interrupt will occur if an attempt is made to index a descriptor that is already indexed.

Typical I/O statement code will start out with the following instructions:

```
MKST

ZERO

NAMC (FILE DESCRIPTOR)

STFF

INDX

LOAD
```

This code will attempt to load word zero of the data pointed to by the file descriptor. The first time this code is executed for a closed file (one which has never been set up) an INVALID OPERAND interrupt will occur. The hardware will take over by calling HARDWAREINTERRUPT. Once in procedure HARDWAREINTERRUPT, it will be determined that the INVALID OPERAND interrupt occurred because we were trying to "INDX" an indexed data descriptor. The size field of seven indicates that this is an unopened file. That is, HARDWAREINTERRUPT is coded to expect the implied open. HARDWAREINTERRUPT calls ATTRIBUTEHANDLER.

ATTRIBUTEHANDLER will allocate a memory area for the File Information Block (FIB) and read in the data pool (see figure 10-3) pointed to by the file descriptor. PCW's for column routines (procedures in the environment of FIBSTACK) are placed in the FIB. These PCWs come from the array IOPCWS which is built by FIBSTACK during initialization. The PCW's for column routines are located in fixed locations in array IOPCWS. A word called SELECTOR (a SIRW) and FIBMSCW (a MSCW) are placed in the FIB. The SELECTOR points to a PCW (PWRITES) in the FIB. The SELECTOR uses pseudo stacks to point to this PCW. The FILESTATE of the FIB is set to NOTOPENSTATE.

The attribute information in the data pool is distributed between two arrays, the File Information Block (FIB) and the Label Equation Block (LEB). The FIB contains PCW's, buffer pointer, pointer to the LEB and other information. The LEB

contains the files title and other attributes.

The original data descriptor (the one pointing to the data pool) is replaced with a descriptor that points to the FIB. The FIB as set up by ATTRIBUTEHANDLER can be seen in figure 10-4. The actual layout of a FIB and LEB can be found in the logical I/O section of the MCP. A PROGRAMDUMP with a file open will help the reader understand this layout.

I/O statement code is generated by a compiler with the assumption that the FIB is set up. If this is the case, when the FIB's descriptor is indexed by 0 and the LOAD is done the SELECTOR will be loaded to the top of stack. If the FIB is not set up, an interrupt will occur.

After the FIB and LEB are set up, an exit is made back to the user program which will index the FIB and load the SELECTOR word.

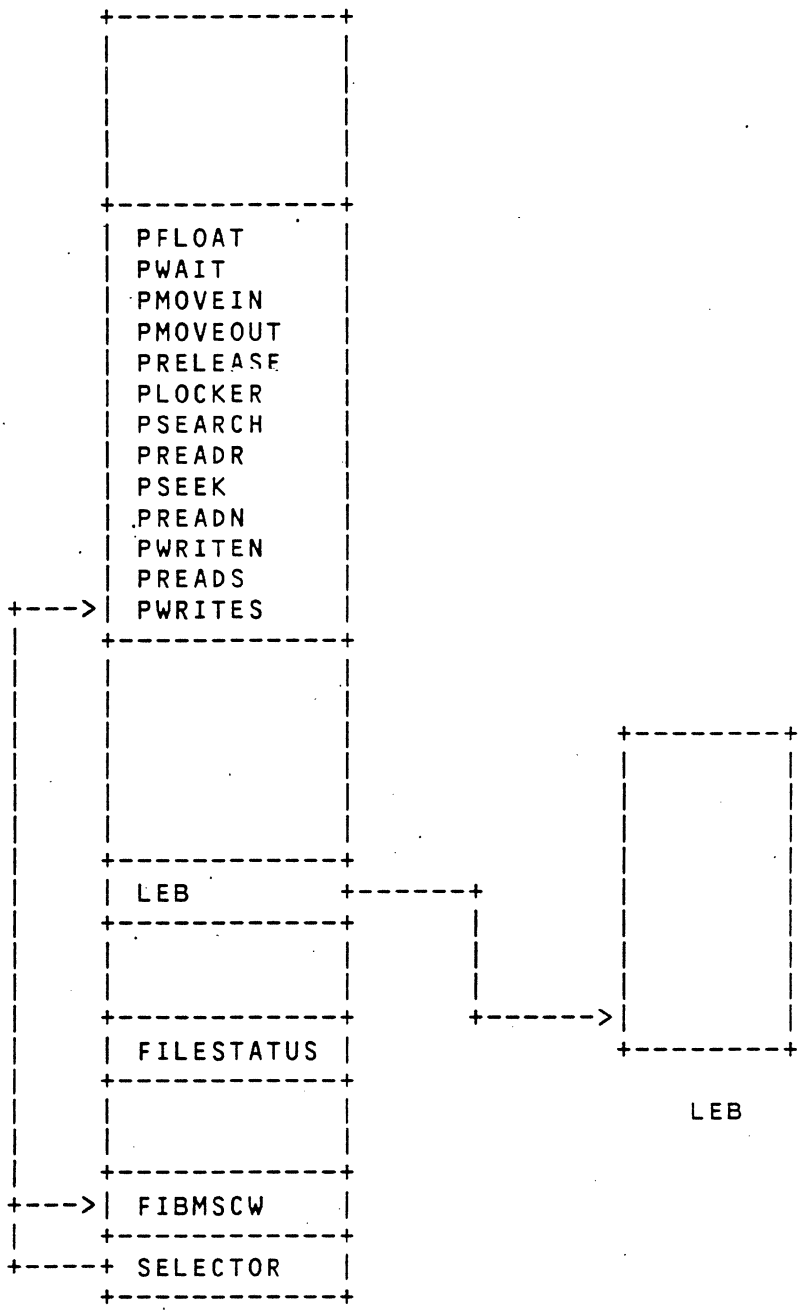


Figure 10-4. FIB as set up by ATTRIBUTEHANDLER

I/O STATEMENT CODE

The SELECTOR word (loaded by the I/O statement code) is used to select an I/O procedure to enter. To better understand how a FIB is entered a closer look at I/O code must be made.

Figure 10-5 contains the in-line code for the I/O statement at the top of the figure. I/O 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. This code will mark the stack and load word zero of the FIB (the SELECTOR SIRW). As stated above this word points to the PWRITES word in the FIB. Thus if an ENTR is done on this SIRW it would be to procedure PWRITES. If a different procedure is needed, the procedures offset is OR'ed into the SELECTOR word. In figure 10-5 a random read is requested. The read random procedure is at offset 3 (from PWRITES) so a 3 is OR'ed into the SELECTOR. This will make the SELECTOR in the top of stack point to PREADN. Thus, for any I/O statement the compiler need only know the offset for the type of I/O and not which procedure (out of 300) to call.

If the file is an Inter-Process Communication (IPC) FIB the SELECTOR will not be set up as described above. An IPC FIB has a SELECTOR which points to the PLOCKER PCW. This is done by placing a 4"17" in the low order bits of the SELECTOR. This means that any offset that is ORed into the SELECTOR will result in the SELECTOR still pointing to PLOCKER. The routine that PLOCKER points to is a lock routine. This routine will make sure that no two stacks are in the same FIB at the same time. After the FIB is locked, the lock routine will call the PCW for the type of I/O the user requested. The requested I/O type is maintained in the CHOOZE parameter.

The second section of code will pass the I/O routine four parameters (see figure 10-6). In general, all I/O routines expect these four parameters. If DIRECT I/O is used the I/O routines expect these four parameters and a reference to an EVENT. When the ENTR is done the procedure selected will be entered and the D1 register will point to the FIB.

After the I/O has been performed, a RETN is made to section 3. The I/O routines return a software result descriptor (RD). The layout of a result descriptor is documented in the I/O subsystem manual (see STATE attribute) If bit 0 is not on in the RD a branch is made to section 5.

If bit 0 is on (section 4) an I/O error has occured and procedure USERIOERROR is called which will DS the program.

Section 5 will delete the RD from the top of the stack.

The I/O statement in figure 10-5 was selected because the ORing of the SELECTOR can be seen. However, the discussion of the program will continue with the WRITE statement.

Because the FIB was just allocated all PCW slots in the FIB contain PCW's to column routines. The WRITE statement in the program will call PWRITES (WRITESCOLUMN). Thus, when the ENTR is done on the SELECTOR it will be to the PWRITES column routine.

```
READ(FYLE[1], 30, EA);

SECTION 1   MKST
            ZERO
            NAMC 2,3            (FYLE)
            STFF
            INDX
            LOAD
            LT8 3               (PROCEDURE OFFSET)
            LOR

SECTION 2   ONE                 (UNITFEATURE, FIRST PARAMETER)
            LT8 1E              (SIZE, SECOND PARAMETER)
            ZERO
            NAMC 2,5            (EA)
            INDX                (AREA, THE THIRD PARAMETER)
            LT48 080000000383 (CHOOZE, THE FOURTH PARAMETER)
            ENTR

SECTION 3   DUPL
            BRFL                (TO SECTION 5)

SECTION 4   NAMC 1,5            (USERIOERROR, DO INTRINSIC)
            EXCH
            IMKS
            NAMC 2,3            (FYLE)
            STFF
            ENTR

SECTION 5   DLET                (DELETE THE RESULT DESCRIPTOR)
```

Figure 10-5. READ Statement Code From FILEEXAMPLE

UNITFEATURE

CONTENTS DEPEND ON "UFCODEF" IN THE CHOOZE PARAMETER. EXAMPLE CONTENTS IS RECORD NUMBER FOR RANDOM I/O's OR CARRIAGE CONTROL INFORMATION.

SIZE

NUMBER OF UNITS (CHARACTERS OR WORDS) TO READ OR WRITE.

AREA

ARRAY USED FOR SOURCE OR DESTINATION OF I/O DATA. PASSED AS AN INDEXED DATA DESCRIPTOR.

CHOOZE

THIS IS A GENERAL PURPOSE PARAMETER WHICH CONTAINS INFORMATION ABOUT THE UNITFEATURE PARAMETER. THE PARAMETER CONTAINS THE TYPE (SERIAL WRITE, SERIAL READ, RANDOM WRITE, RANDOM READ) OF I/O, THE LANGUAGE (ALGOL, COBOL, FORTRAN) AND OTHER INFORMATION (SPACE, SKIP, STATION, STACKER).

Figure 10-6. Parameters Passed for I/O Operations

COLUMN ROUTINES

The column routines are called when the FIB needs to change state. They will decide what to do based on the FILESTATE of the FIB. Because the FILESTATE is NOTOPENSTATE, procedure OPEN (FIBOPEN) is called.

OPEN will assign the logical file (as represented by the FIB) to a physical file (as represented by a DISK file). OPEN will call FINDOUTPUT to assign output files and FINDINPUT to assign input files.

OPEN will also call procedure SETUPTANKP which is responsible for setting up all Input Output Control Blocks (IOCB) and buffers. Each buffer contains a fixed number of words for buffer control and an area for user data. The fixed part contains a descriptor that points to the IOCB, descriptor that points to the user data, event for I/O finish, an I/O Control Word (IOCW) and a link to the next buffer. The link word is used when buffers are rotated. The AREADESC word in the IOCB points to the user data area. The IOCB word of a FIB points to the current IOCB and the BUFFDESC word of the FIB points to the current user data area. The IOAREA word of the FIB points to the current buffer. The buffers are shown in figure 10-7. The FILESTATE is set to NULLSTATE.

After the call on OPEN the PWRITES PCW is called again. In most cases this calls WRITESCOLUMN again. Because the FILESTATE is NULLSTATE, INITIALIZEFIB (SETUPFIB) is called. This procedure will select PCW's for diagonal routines for the FIB. The FILESTATE is changed to WRITESSTATE. The other code in a column routine is for normal (READ to WRITE) FIB state changes.

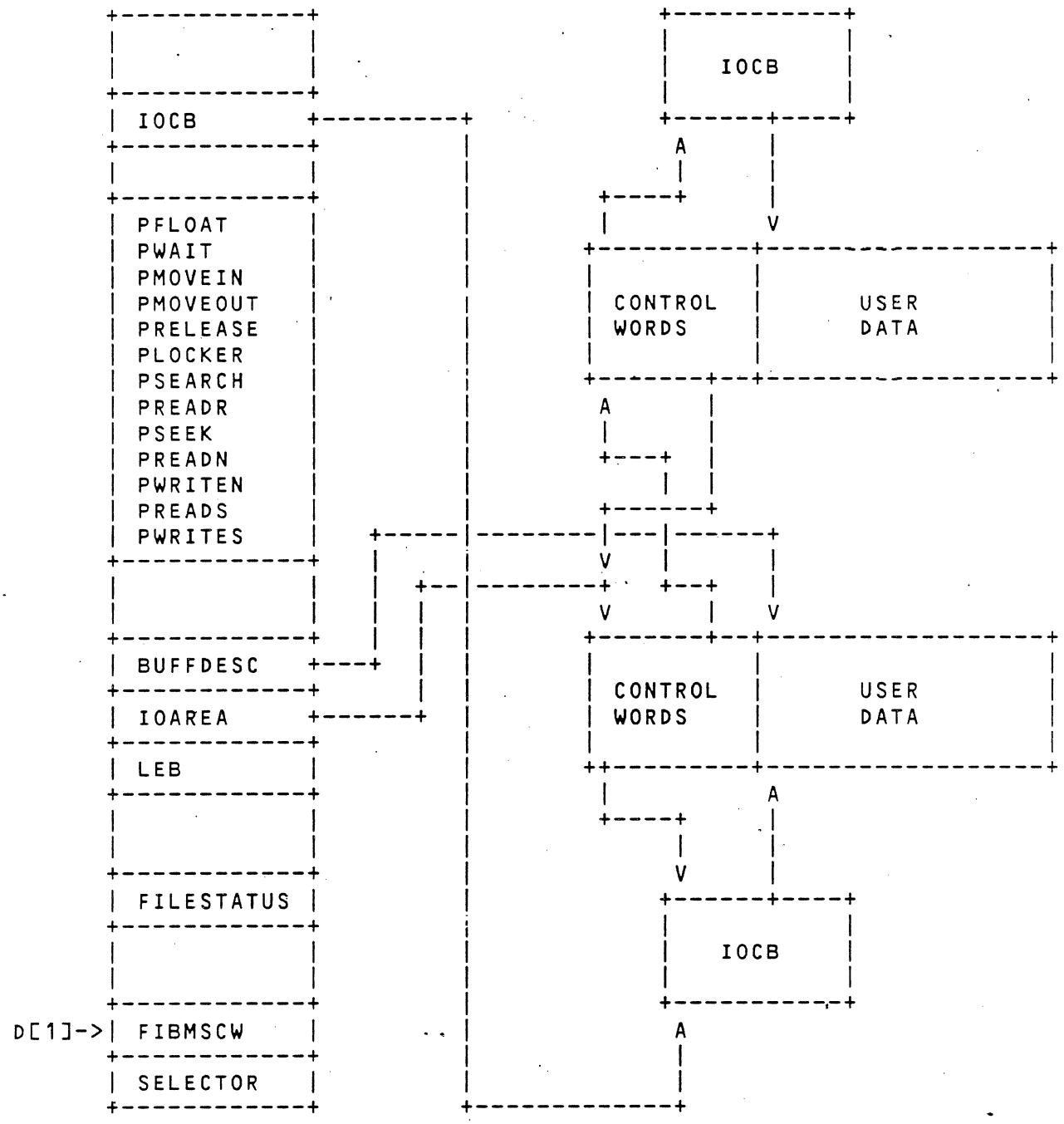
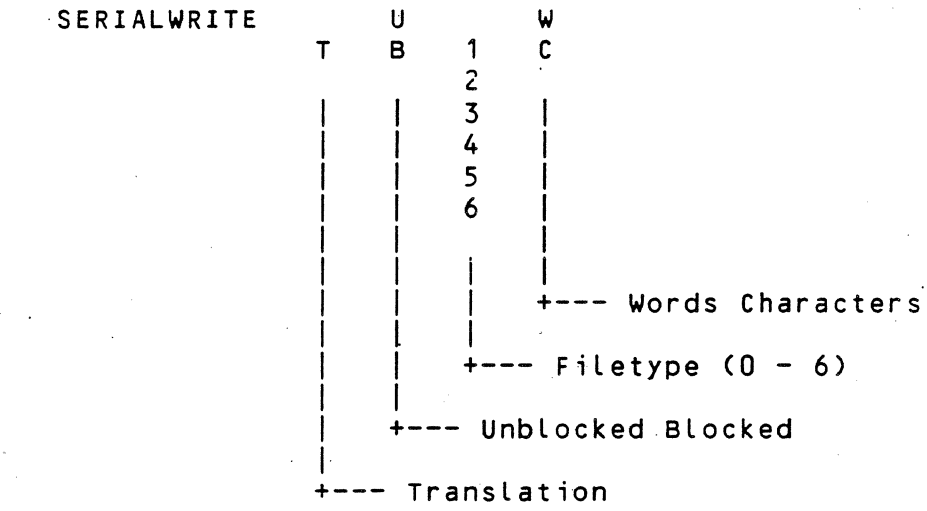


Figure 10-7. File Set Up By OPEN

PCW SELECTION

Since I/O requests can occur for many different file types and record formats the FIBSTACK implementaion must be designed to avoid excessive decision making. This is accomplished by providing complete sets of diagonals for each decision category. The categories are word or character, blocked or unblocked, file type and software translation. For example, there is a routine to handle a serial read for a word oriented, blocked, type 3, file.

The diagonals follow a naming convention in FIBSTACK. This convention is: basic diagonal, translation, blocking, file type and frame size. For example, the serial write basic diagonal can be modified as follows:



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 10-8. The first section consists of general PCW's such as column routines, OPEN, CLOSE, ILLEGALCOLUMNACTION and other procedures.

ILLEGALCOLUMNACTION in called the BAD PCW in FIBSTACK. This PCW is placed in a PCW slot in the FIB if the I/O operation is not valid. For example, it would be placed in the WRITE PCW slots of a card reader file. ILLEGALCOLUMNACTION will result in the program being DS'ed.

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 all other sections. This define references the PCWCONTROL in the associated FIB.

SETUPFIB uses information in the FIB to select diagonals. The diagonals selected are based on the categories described above. SETUPFIB builds PCW's for 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. This word is referenced as follows:

P0LOC = PCWCONTROL.[15:16]

Always points to a serial write PCW.

P1LOC = P0LOC + 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.

P0LOC 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.

The PCW's currently in the PWRITES, PREADS, PWRITEN and PREADN FIB locations are COLUMN PCW's. SETUPFIB has selected the proper PCW's based on the type of the file. The last thing SETUPFIB does is to fill the buffers and then return to WRITESCOLUMN.

WRITESCOLUMN will exit the CASE statement, set the FILESTATE to WRITESSTATE and replace the column PCW in PWRITES with a diagonal PCW. Thus, the PWRITES PCW is overwritten with IOPCWS[P0LOC]. If you recall, P0LOC is a define referencing the PCWCONTROL word in the FIB. The PWRITES PCW slot now contains a PCW to SERIALWRITEBW (SERIAL WRITE BLOCKED WORDS).

The column routine calls PWRITES again. This time it calls SERIALWRITEBW. After the data is transferred to the users

array a return is made to the user program.

From this point the state of the file will determine which diagional PCW is installed in the FIB.

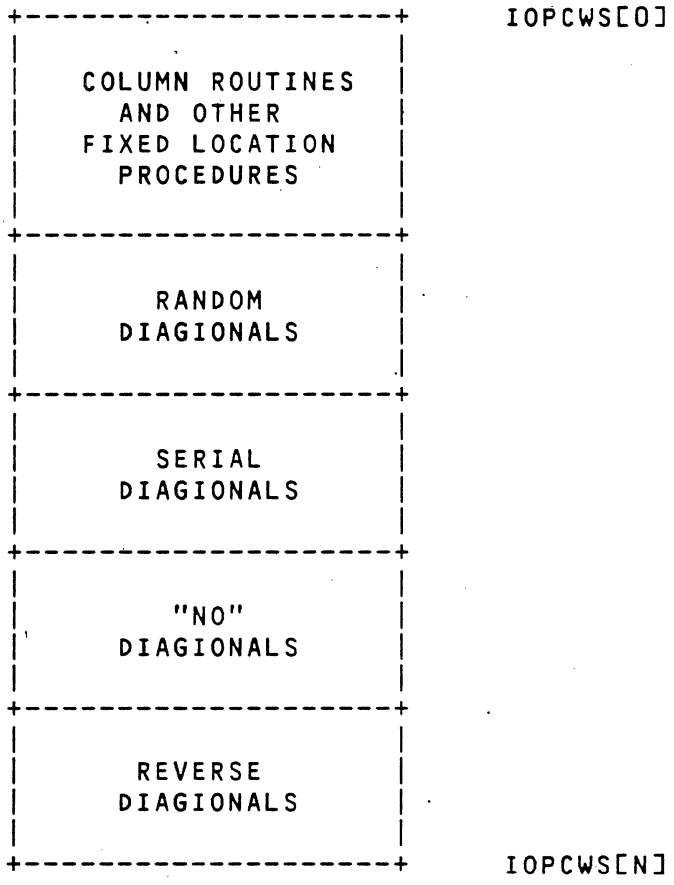


Figure 10-8. IOPCWS Array Layout

I/O PROCEDURE FLOW

The FILESTATE of the FIB is now WRITESSTATE. SERIALWRITEBW has been called to do a logical I/O. This procedure has six statements (defines) which transfer data from the EA array to the buffer and return to the user program.

The first statement in the diagonal (SERIALWRITEBW) is WAITB. This define checks to see if the current buffer is a new buffer. If it is a new buffer, a buffer rotation has taken place and the routine must be sure the buffer is not still in process. That is, we must be sure an I/O is not in process for the buffer. The PWAIT PCW is called (WAITONBUFFER statement) to check the buffer status. PWAIT contains a PCW to WAITDISK. Procedure WAITDISK will change the ADDRESS in the BUFFDESC word in the FIB to point to the new buffer. The procedure will then wait on the event in the buffer (fixed words in the buffer) pointed to by the IOAREA word in the FIB. This event is called BFFREVENT in the buffer. When the event has happened, the buffer has been transferred to disk and the user may reuse the buffer. This buffer is now pointed to by the BUFFDESC word of the FIB.

The next statement in SERIALWRITEBW is MINIMIZER. This define will check to see if the user is requesting an I/O larger than the MAXRECSIZE. If this is the case, the requested size is set to MAXRECSIZE.

The next statement in this procedure is MOVE. This define moves the data from the users array to the buffer. The buffer is pointed to by the BUFFDESC word and indexed by the OFFSET (current offset in the block) word.

The next statement in SERIALWRITEBW is UNBLOCKB. This define adds the record size to the current record offset and checks to see if the current offset in the block is greater than or equal to the size of the block. If it is, the buffer is released (RELEASEBUFFER). RELEASEBUFFER is a call on the PRELEASE PCW. The PRELEASE PCW contains a PCW to procedure RELEASESERIALDISK. This procedure will reset the I/O complete bit in the buffer and initiate the I/O. When the I/O is complete IOFINISH will cause the BUFFEREVENT in the IOCB which will set the I/O complete bit and wake up any tasks that were waiting on the I/O to complete. After the I/O has been initiated buffers are rotated. Buffer rotation uses the IOAL word in the current buffer to get a pointer to the next buffer. The IOAREA word is overwritten to point to the new buffer and the IOCB word is overwritten to point to the new IOCB. The BUFFDESC word is not overwritten at this time so it points to the old buffer. The FIB new buffer bit is set so the WAITB define will overwrite the BUFFDESC word.

The last statement in the diagonal routine is FINISH. This define will return the result descriptor to the user.

When an I/O is done which requires data to be written to or read from disk, the disk header will be used. The disk header is pointed to by the DHEADER word in the FIB. This word is an SIRW which points to the DISK FILE HEADER STACK location.

On all subsequent writes, the diagonal procedure will be entered. The diagonal PCW will not be changed unless the file changes state. This could happen if the user does anything other than the state that the file is currently in (WRITESSTATE). If another type of I/O is done a column routine will be entered which will change the PWRITES diagonal to a column routine and insert the new diagonal in the proper PCW slot. At any point in time there is only one diagonal routine in the FIB, all other PCW's are column routines.

When the program ends, the disk file will be closed by BLOCKEXIT. At this time, the the header will be written into the FLAT DIRECTORY (if it is a permanent file), FORGETSPACE will be called for the HEADER and buffers. If the file were simply closed the FIB and LEB would remain in memory. However, because a BLOCKEXIT close is performed the FIB and LEB will be released.

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 NOREADBW

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.

THIS PAGE INTENTIONALLY LEFT BLANK FOR FORMATTING PURPOSES

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS