THE MEMORY MANAGEMENT FUNCTION IN A MULTIPROCESSOR COMPUTER SYSTEM - A Description of the BCC 500 Memory Manager

Wrenwick K. Lee

Technical Report R-2
Issued September 12, 1974

THE ALOHA SYSTEM, Task II
Department of Electrical Engineering
University of Hawaii

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 1956
Contract No. NAS2-6700

THE ALOHA SYSTEM, Task II, is affiliated with the Department of Electrical Engineering at the University of Hawaii, and is currently conducting studies into the design and fabrication of secure multiprocessor operating systems.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by NASA, Ames Research Center under Contract No. NAS2-6700.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency of the United States Government.

ABSTRACT

The BCC 500 System was designed to be a large-scale interactive computing utility supporting up to several hundred on-line users. To achieve this goal, a multi-processor architecture was chosen. Central processors were designed to compile and run user code, while the operating system was designed to be run by several dedicated processors. This paper details the system memory management function which was assigned to one of these processors.

With this architecture, memory management takes on a different dimension. Techniques and capabilities beyond the reach of those attainable by time-sharing memory managment functions on a CPU with other system and user tasks may be employed. The memory manager is continuously active, monitoring the memory system and taking appropriate action more quickly and more often.

ACKNOWLEDGEMENTS

The implementation of the BCC 500 memory manager involved the contributions of various members of the BCC 500 staff, among them Jack Freeman, Dr. Butler Lampson, Dr. W. Lichtenberger, Dr. Melvin Pirtle, Rainer Schulz, and Robert Van Tuyl.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Organization 1

1.1.1 System Structures (Section 2). 1

1.1.2 Hardware (Section 3) 1

1.1.3 Data Structures (Section 4) 1

1.1.4 Microcode (Section 5 and Section 6) 2

1.1.5 APU Code (Section 5 and Section 7) 2

1.1.6 Communications, Error Handling (Section 8) 2

1.1.7 Concluding Remarks (Section 9) 2

2. INTRODUCTION - SYSTEM STRUCTURES

2.1 Pages 3

2.2 Page Types 4

2.3 Unique Names 4

2.4 Physical Location of Pages 8

2.5 Context Blocks 11

2.5.1 The Process Memory Table 12

2.5.2 The Active Page Table 17

2.6 The Process Table 19

3. INTRODUCTION - THE HARDWARE

3.1 The Processor 25

3.2 Drum & Disk Hardware 26

3.2.1 Data & Address Formats 26

3.2.2 Position Counters 39

3.2.3 TSU Control Registers 40

3.2.4 Indicators 45

3.2.5 TSU Instructions 47

3.2.6 Instruction Timing 50

3.2.7 Status Register 51

3.2.8 Attentions 54

3.2.9 Select Register in the TSU 54

3.2.10 Disk Seeks & Position Verification 57

3.2.11 TSU State 59

4. INTRODUCTION - DATA STRUCTURES

4.1 Queues 61

4.1.1 General Request 66

4.1.2 Activate Request 67

4.1.3 Request Entry 68

4.2 The Core Hash Table 70

4.2.1 CHT Layout 70

4.2.2 When a Page is 'in CHT' 71

4.2.3 CHT Page Status Information 74

4.2.4 Scheduled Count & Accessibility of Pages 75

4.2.5 Removing a Page from CHT, or Clearing DIRTY 77

4.3 Drum Hash Table 79

4.3.1 General 79

4.3.2 DHT Page Status Information 80

4.4 Standard Circular List Structure 82

4.5 State of AMC Microprocessor 84

4.6 Important Central Memory Locations 84

4.7 Statistics (Counters) 87

4.8 Table Manipulations 90

5. INTRODUCTION - SOFTWARE & FIRMWARE 95

6. MEMORY MANAGER FIRMWARE

6.1 The Memory Manager's Main Loop 99

6.2 Microcode Descriptions 100

6.3 Detailed Microcoded Routines Descriptions 104

6.3.1 Subroutine Linkage 105

6.3.2 Main Loop 105

6.3.3 Process Attention 107

6.3.4 Start Auxiliary Processing Unit 109

6.3.5 CHT Hash 109

6.3.6 CHT Search 109

6.3.7 Enter CHT Entry 110

6.3.8 Delete CHT Entry 111

6.3.9 Clear CHT Entry 111

6.3.10 Get Free Core 112

6.3.11 Put Page on Free Core List 113

6.3.12 DHT Hash 113

6.3.13 Search DHT 114

6.3.14 Make DHT Entry 115
6.3.15 Delete DHT Entry 115
6.3.16 Append Entry onto List 117
6.3.17 Stack Entry on List 118
6.3.18 Remove Entry from List 118
6.3.19 Save State 119
6.3.20 Dump TSU State 119
6.3.21 Generate Wakeup 120
6.3.22 Send TSU Instruction 121
6.3.23 Stack Entry on Free List 121
6.3.24 Remove Entry from Free List 122
6.3.25 Initialization Sequence for AMC 122
6.3.26 Wait Until Device Idle 124
6.3.27 Compute Next Sector on Selected Unit 125
6.3.28 Get Position of Rotating Device 126

7. MEMORY MANAGER SOFTWARE

7.1 APU Code - Overview 127
7.2 Major Transfer Vectors 131
7.2.1 Primary Transfer Vector 132
7.2.2 General Request Transfer Vector 132
7.2.3 Activate Request Transfer Vector 133
7.2.4 Cleanup Request Transfer Vector 134
7.2.5 Startup Request Transfer Vector 135
7.3 The Auxiliary Processing Unit 136

8. COMMUNICATIONS AND ERROR HANDLING

8.1 Communications - The Memory Manager and the Outside World. 148

8.2 Error Handling 150

8.2.1 General Error Philosophy 150

8.2.2 Types of Errors. 151

8.2.3 Errors During Swapping 152

8.2.4 Disk Read Errors 152

8.2.5 Disk Write Errors 152

8.2.6 Drum Errors for Non-Swapping Transfers 153

9. CONCLUDING REMARKS 154

Appendix I Abbreviations for Model I Memory Management System . . 157

Appendix II Microprocessor General Theory of Operation 160

Appendix III AMC Startup 180

LIST OF FIGURES

Fig. 2.1 MIB and Associated Page Types CB, IB, FP 5

Fig. 2.2 CB Contents 13

Fig. 2.3 CB Storage Allocation 14

Fig. 2.4 Format of an Entry in a Process Memory Table 20

Fig. 2.5 Format of an Entry in an Active Page Table 21

Fig. 2.6 Format of the Process Table (PRT). 23

Fig. 2.7 Description of PRT Bits 24

Fig. 3.1 Auxiliary Memory System Configuration 27

Fig. 3.2 Drum and Disk Page Formats 28

Fig. 3.3a Drum Characteristics 31

Fig. 3.3b Disk 32

Fig. 3.3c Disk (continuation) 33

Fig. 3.4 Parameters of BCC 500 Rotating Memories 35

Fig. 3.5a TSU Control Registers 37

Fig. 3.5b TSU Registers 38

Fig. 3.6a Position Counter (PC) Format 41

Fig. 3.6b Drum Record Timing 42

Fig. 3.7 TSU Register Loading 44

Fig. 3.8 TSU Register Loading Interface (Select Register) 55

Fig. 4.1 Normal Request Entry 62

List of Figures - continued

Fig. 4.2 Various AMC Queues 63

Fig. 4.3 Direct I/O Request Entry 70

Fig. 4.4 Core Hash Table (CHT2) Entry 72

Fig. 4.5 CHT Hashing Structure 73

Fig. 4.6 Drum Hash Table Entry 81

Fig. 4.7a Circular List Structure 83

Fig. 4.7b Empty List 83

Fig. 5.1 AMC Code Organization 96

Fig. 6.1 Main Loop 98

Fig. 6.2a Three Hash Table Entries 116

Fig. 6.2b b is Deleted 116

Fig. 6.2c Corrected Hash Table if a, b, c Hash into same Location. 116

Fig. 6.2d Corrected Hash Table if c Hashes into Location it is in. 117

Fig. 7.1 Dispatching Structure 128

Fig. 8.1 Memory Manager Communications 149

APPENDIX

Fig. 1 Microprocessor Data Paths: Arithmetic Section 161

Fig. 2 Microprocessor Data Paths: Control Section 162

LIST OF TABLES

Table A1. 90-bit Microinstruction Word 168

Table A2. Branch Conditions 172

Table A3. Special Functions 176

Table A4. Bool Box Control 179

1 INTRODUCTION

Memory management for the BCC 500 was designed to be controlled by a single dedicated processor. As a result, the central processors do not run memory management tasks. The memory management processor continuously monitors the memory system and quickly responds to events that are important to the memory system.

This report is a technical description of the memory management system. Hardware components, data structures, implementation code, and communications conventions are described in detail.

1.1 Organization

The following parts of the memory management system will be described.

1.1.1 System Structures (Section 2)

This describes the system environment in which the memory manager operates. Concepts such as page, unique name, context block, process table, etc. are discussed.

1.1.2 Hardware (Section 3)

The hardware has been specially adapted to attain the high swap rates desired. The processor, drum, disk and auxilary memory controller are extensively discussed. One can see the differences such as position monitoring, unique name checking, and other features of the hardware.

1.1.3 Data Structures (Section 4)

The fundamental data structures of the memory manager are

detailed. The queue which is used extensively in the memory manager is illustrated. Also the core and drum page tables are explained.

1.1.4 Microcode (Section 5 & Section 6)

The role of microcode is explained. Various examples of microcoded routines are given.

1.1.5 APU Code (Section 5 & Section 7)

The role of APU code is explained. The APU instruction set is given.

1.1.6 Communications, Error Handling (Section 8)

Communications, and error conventions are explained.

1.1.7 Concluding Remarks (Section 9)

Looking in retrospect, comments are given on the relative merits and difficulties of the implementation. Recommendations and questions are also mentioned.

2 INTRODUCTION - SYSTEM STRUCTURES

The following are the important system structures involving the memory manager. An overview of the various structures involved in paging is given and one is introduced to the environment of the memory manager. The number of structures presented is essentially quite small, yet sufficient to give a fair insight into the role of the memory manager.

2.1 Pages

The memory system of the BCC 500 consists of 128K words of core, 4 million words of drum, and 125 million words of disk. The BCC 500 System super-imposes a page structure on this storage space. All three levels of storage are sub-divided into 2K-word blocks, called pages. Pages are units of information as well as units of storage space. When we speak of pages of code, pages of data, etc., we mean an amount of code, data, etc., that may be stored in a page of storage. This is just to say that "page" is used in a manner completely analogous to that in which "byte" and "word" are used. When we use "page" to refer to a unit of storage space, we speak of "core pages", "drum pages", and "disk pages" depending on which of the three levels of memory we are referring to. Storage pages have an "origin" as well as an extent (2048 words). Pages of core are 2048 word blocks starting at an address which is congruent to 0 modulo 2048. Similarly, drum and disk pages have fixed "starting addresses" built into the hardware. They are a little different from core pages in that we don't speak of word addresses in connection with these storage devices.

2.2 Page Types

Pages can be of the following types:

1) MIB (Multiple Index Block) - the file directory for one user. It also contains the disk addresses of the data pages for small files.
2) IB (Index Block) - contains the disk addresses of the data pages for a large file.
3) CB (Context Block) - contains the defining information for a process.
4) FP (File Page) - contains data and occupies a definite position in a file.

The pages of a file are numbered from 0, up to a maximum value MAXFA. The number for a page is called its file address and specifies its position in the file, which may not be the same as the number of pages preceding it, since some file addresses may have no corresponding pages.

5) PP (Private Page) - contains data and belongs to a process. Private pages are not ordered in any way, but each one can appear in just one PMT* byte.

The relationships of the various types of pages are shown in Fig. 2.1.

2.3 Unique Names

Associated with every page is a 48-bit number called its <u>unique name</u>. This number has the following format:

---

*PMT is a table of pages belonging to a given process. This is explained further in Section 2.5.1 on the PMT.

MIB and Associated Page Types CB, IB, FP

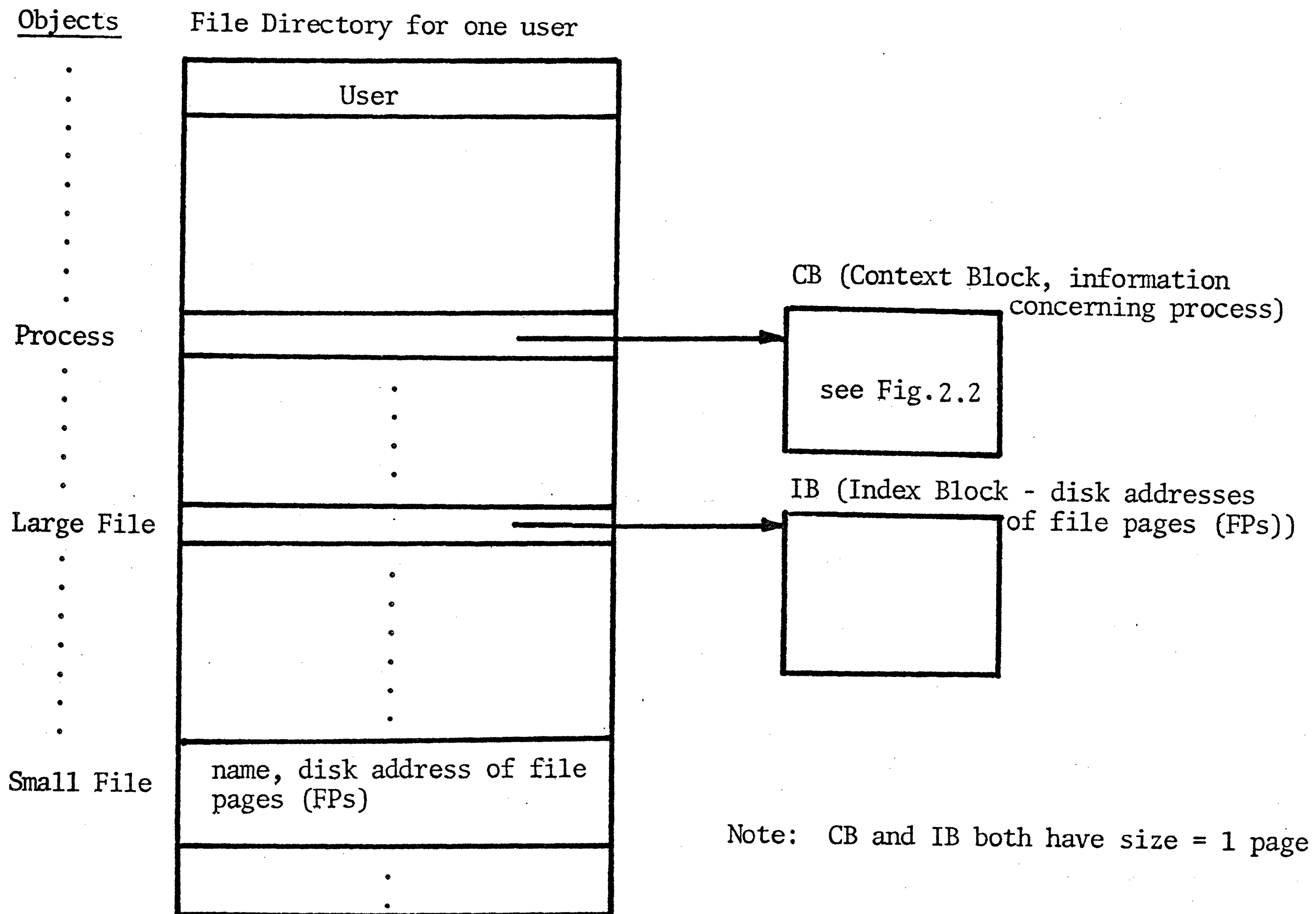


Figure 2.1

Bits	Name	Contents
0-1	UNTAG	0 = MIB or small file data page
		1 = large file IB or data page (FP)
		2 = CB or PP
		3 = not used
2-17	UNUSER	User number of the owner of the page
18-36	UNID	A number different for each object (file or process) with the same owner. For a MIB this field is 0.
37-47	UNADR	Page number + 1 for a private page or a file page. For a MIB, IB or CB this field is 0.

Some comments on the implications of this format are appropriate.

1) The number of distinct users is limited to 64K. Each user who exists has exactly one MIB. Every page except a MIB belongs to a MIB, and hence every page belongs to some user. The user to whom a page belongs can be determined from the UNUSER field of the page's unique name.

2) Every object (file or process) has an entry in a MIB. This entry specifies the UNID field of the object. This field is the same in the unique name of every page of the object. It is therefore possible to find the object which a given page is part of, starting from the unique name of the page, by examining the UNUSER field to find the MIB and then comparing the UNID field with each object in the MIB. When a new object is created it is given a new UNID, which is the UNID of the last object created in that MIB, + 1. This ensures that all pages of the new object will be distinguishable from any

page of any old object, whether or not the old object still exists. When the last UNID value (2^{19}-1) is assigned to a new object in some MIB, it is not possible to create any more objects in that MIB. This situation can be remedied only by reassigning the UNIDs for all the presently existing objects in that MIB and updating all occurrences of unique names for pages belonging to that MIB anywhere in the system.

3) A file cannot have more than 2047 pages because of the size of the UNADR. It is not possible to move a page to another position in a file without changing its unique name. It is, however, possible to completely reconstruct a file from a complete scan of the disk.

Unique names have the following basic property: no two different pages ever created by the system can have the same unique name. This observation requires some clarification:

1) If the UNID reassignment mentioned above is carried out, it must be thought of as the creation of a new system into which some information from the old system is copied.

2) If a file page is destroyed, and a page with the same file address in the same file is subsequently created, it is considered to be the same page. There is no difference between these two operations performed in sequence and the operation of zeroing the page. It is not claimed that this is the only possible interpretation of the meaning of recreating a file page, but it is a reasonable one, and it is the one adopted by the Memory Management System (MMS).

3) It is not possible to have more than one reference to a private page When the page is destroyed it is removed from all maps in the process

and therefore ceases to exist in the most complete sense possible. If another private page is created in the same PMT byte, it makes no whether it is regarded as the same page or different (though the latter viewpoint seems preferable):

--if it is the same, the matter is academic since there can be no references to the page at the time it is created and subsequent references to it are under the control of the creator.

--if it is different, the matter is academic since no record of the previous page exists anywhere in the system.

Unique names exist in the system in the following places:

1) Recorded on disk together with a page
2) Recorded on drum together with a page
3) In the core hash table (CHT) entry for a page
4) In the entry for an object (and implicitly for each of its pages) in a MIB
5) In a PMT entry
6) In a PRT* entry

They also exist in the temporary storage of the MMS.

2.4 Physical Location of Pages

A page always exists on disk, and may also exist on drum, in core, both or neither, where by 'exist' we mean that space is allocated for the page. A valid copy of the page may exist in any combination of these three places.

---

*PRT is the Process Table -- This is explained further in Section 2.6.

Since a page p always exists on disk, there is always a disk address K(p) for it. This address is assigned by the monitor, which is responsible for disk allocation, when the page is created. The unique name recorded at K(p) is either UN(p) or 0, the latter if and only if the WUN (Write Unique Name) bit for the page is set in the Drum Hash Table (DHT). The reason for this is that when the page is created it is assigned a free disk page which has UN = 0 and the WUN bit for the page is set. The first time the page is written on the disk, its unique name will be written and the WUN bit will be cleared. When p is destroyed the unique name at K(p) is zeroed.

It is expected that pages will be moved on the disk infrequently, if at all. The combination of unique name and disk address is therefore normally sufficient to identify a page and permit it to be read from the disk; this combination is called the real name (RN) of the page and occupies three words. If a page is moved and an attempt is made to access it using the old disk address, a unique name error will occur, since it is not possible for the same unique name to be written on two different disk pages. The unique name of the page can then be used to find the MIB entry for the object of which it is a part, and from this entry the correct disk address can be obtained. This works because of a second basic property of the system: <u>the correct disk address of a page is always recorded in the MIB entry (for small file page), IB (for large file page) or CB (for private page) for the object of which it is a part</u>. Hereafter we will suppress the IB and CB cases.

It is possible for many occurrences of the real name of a page (other than a private page) to exist in the system. New occurrences are created when a file page is put into a PMT entry, or when the monitor puts a MIB, IB

or CB into its map, or when a PRT entry is made for a process.

The subset of pages in core or on the drum changes rapidly, as does the physical core or drum location of a page. Core and drum addresses are therefore kept in exactly one place in the system, so that only one change need be made when a page is moved.

1) The core address of a page is kept in the Core Hash Table (CHT), which can be conveniently accessed by unique name. Every page which is in core (the precise meaning of this phrase will be explained later) has a CHT entry, which contains all the information relevant to its sojourn in core.

2) The drum address of a page is kept in the Drum Hash Table (DHT), which in theory can also be conveniently accessed by unique name. Every page which is on the drum or on its way there has a DHT entry, which contains all the information relevant to its sojourn on the drum.

We are now in a position to explain roughly how a page p can be accessed if its RN is known. First use UN(p) to access CHT. If this access is successful the page is in core at the address given by CHT. Otherwise use UN(p) to access DHT. If this access is successful the page is on the drum at the address given by DHT. Otherwise use K(p) to access the disk. If the unique name recorded there agrees with UN(p), the page is there on the disk. Otherwise use UN(p) to find the MIB entry for the page. This may reveal that the page does not exist. Otherwise it is on the disk at the location given by the MIB, and the RN may be updated accordingly.

It is a third basic property of the system that <u>the procedure described above will always yield a valid copy of the page unless there is a failure of</u>

the MMS. There are some built-in cross-checks to make detection of a failure more likely.

1) When a page is read from drum, the UN which was recorded on the drum may fail to agree with DHT. We give up this cross-check by accessing DHT on disk address rather than unique name. Since it is possible for a disk address to be reassigned (unlike a unique name), an attempt to access a page p on the drum with an old RN r (i.e., $K(p) = K(r)$) may fail because $UN(r) \neq UN(p)$. We can, however determine the facts. If the UN recorded at K(p) agrees with the one on the drum, then we have an old RN and the MMS has not failed. If it disagrees there is indeed a MMS failure. To summarize, using UN(p) to access a DHT keyed on disk address must include reading the page from the drum and checking the recorded UN.

2) When a page is read from disk using a disk address obtained from the MIB, the recorded UN must agree with the one in the MIB.

3) When writing on the disk, the recorded UN must agree with the one obtained from DHT and the drum. This cross-check of the (UN,K) pairing defined by DHT and the drum is gained in return for cross-check (1) which was lost by keying DHT on K.

2.5 Context Blocks (CB)

To define a process for the operating system requires a good deal of information. This information is called the "state" of the process. When a process is dormant, its state is defined by its entry in its owner's file directory (MIB). Such an entry contains the symbolic name of the process, information for controlling access to the process, and the Unique Name of a special

page of the process called its Context Block. This special page contains the information needed to introduce (or re-introduce) the process into the operating system's job stream, that is, to activate the process.

When a process is active, its state is more complex. Some information about it is kept in tables, such as the Process Table and the character I/O line tables, which are resident in core. Information which is needed only when the process is itself in core (or being swapped in or out) is kept in the Context Block page. This page can be thought of as providing temporary storage for the operating system in certain of its functions with respect to the process.

Figures 2.2 and 2.3 detail the layout of the CB. Of particular interest are the MAP, PMT and APT which involve the memory manager.

2.5.1 The Process Memory Table (PMT)

One kind of information the operating system requires about an active process is a list of the Unique Names of all the pages which belong to the process. These names (together with a mapping of the process' address space into them) are needed by the CPU so that it may find the pages to which the process directs references when actually executing instructions. These page names are also required by the Auxiliary Memory Control (AMC) so that it can identify the pages which it needs to swap into core preparatory to the running of the process.

The page names, and some additional information about the pages, are kept in a table called the Process Memory Table (PMT) in the Context Block (CB). These tables begin in a standard place (loc. 300_8) in each process' Context Block for the convenience of the various parts of the operating

CB CONTENTS

0:	POP entry indirect address word
1:	POP entry indirect address word
2:	SP first unused stack address
3:	SL last word allocated for stack
4:	P for Trap (ring dependent)
5:	PAR for Trap (ring dependent)
6:	BRU for Trap (ring dependent)
7:	BRU for Trap (ring dependent)
10-177:	FREE
200-277:	MAP - Map associating virtual addresses with pages (contains indices into PMT)
300-1277:	PMT - Table of pages in use by this process
1300-1707:	SPT
1710-2027:	SPCS
2030-2115:	ICT
2116-2235:	OFT
2236-2650:	STACK
2659-2751:	APT - Table of active pages (contains indices into PMT)
2752-1763:	TRSTATE
2764-2775:	SWSTATE
2776:	CTC
2777:	IT

Figure 2.2

CB STORAGE ALLOCATION

SPT (42 words/entry)	264
OFT (5 words/entry)	80
PMT (4 words/entry) (128)	512
APT (1 word/entry)	65
SPCS (5 words/entry)	80
	1011
MAP	64
STATE, ETC.	25
	1090

Figure 2.3

system which must reference them. They have room for 128 page names, but can be expanded to allow for up to 255. That is, the limit of 255 is built into the system in a number of places, but the current 128 page limit is imposed by only the software part of the system. We begin by giving explanations of the contents of entries in PMT. Refer to Fig. 2.4 for a picture of a PMT entry.

Unique Name: These two words hold the Unique Name of a page of information. This is the same Unique Name as is written with the page on the disk and drum and kept in the Core Hash Table when the page is in core. It is used by the CPU's map loader when it looks up the page in CHT and by the Swapper when it is swapping the process in or out.

Disk Address: This field holds the address at which the disk copy of the page is stored. It is the address which will actually be sent to the disk TSU (Transfer Sub-Unit) when it is required to read the page into core or to write it on the disk. Such addresses have to be kept because there is no provision at the TSU level for addressing pages by their Unique Names. However, the system does not depend on this disk address being correct. When the transfer of a page to or from the disk begins, the contents of the Class Code field of the addressed page is checked for equality with the Unique Name of the page of information it is desired to transfer. If this check fails the transfer is aborted and a "Class Code Error" is reported to the process for which the transfer was being done. A page's Unique Name and Disk Address are called together its "Real Name."

This is a good place to note an implementation concession for the Drum Hash Table. First we note that the Core Hash Table is a table entered by hashing the Unique Name of a page and containing for each entry the Real Name

of a page and the absolute address of the core page in which the page is currently stored. Ideally the Drum Hash Table would be completely analogous and each entry would contain a Real Name and a drum page address of the current drum copy of the page. This implementation was not possible, simply because of the amount of core storage which such a table would require. Instead, DHT entries contain only the Disk Address word of the Real Name. Except for the loss in elegance this seldom causes any problems. It just means that in certain cases we have to do an otherwise unnecessary read from the drum to compare a Class Code with a Unique Name.

(The Core Hash Table and Drum Hash Table are explained in greater detail in the section on Data Structures.)

Thus the Disk Address word in PMT entries is used to find the page whose Unique Name appears in the entry both on the disk and on the drum, but in neither case is it considered the final authority in the matter since we always make the comparison between Class Code and Unique Name.

RF (Reference Flag): The system tries to make sure that the pages in the Core and Drum Working Sets of a process are the ones that the process is referencing most frequently. In order to do this, it must somehow be kept informed as to what pages the process is referencing. The CPU's Map Loader provides this information by setting the RF flag in PMT whenever it loads the corresponding page into its map.

SF (Scheduled Flag): When a program causes the Map Loader to load a page into the CPU's map. the Map Loader looks up the page in the Core Hash Table using the Unique Name in the appropriate PMT entry. It is possible that the page is in core for some other process but not supposed to be

available to the process in which the program is running. Giving the program access to the page under these conditions will in general lead to chaos, since the core storage management system depends on knowing how many processes have access to the pages in core. The SF bit is used to prevent this illegal access. It is set by the core management system if the process is authorized to access the page, and the CPU will trap if it is asked to load a page with SF = 0 into its map.

CCE (Class Code Error): When the pages of a process' Core Working Set are being read into core the Unique Names in PMT are compared with the Class Codes on the pages read. If the comparison fails, the read is aborted and the CCE flag in the PMT entry is set. The SF bit is, of course, reset. If the process tries to reference the page, it will get a trap from the CPU, at which time CCE can be tested to determine the source of the problem.

RDE (Read Error): If a page of the Core Working Set cannot be swapped in because of a "hard" error encountered in trying to read the page from the drum, the RDE bit in the pages PMT entry is set. Like the CCE bit, this error indication is for use in analyzing the page fault which will result if the page is referenced.

2.5.2 The Active Page Table (APT)

When it is time to bring a process into core so that it may execute instructions on a CPU, a request is sent to the Swapper to bring in the pages the process needs in order to run. The Swapper is given a pointer to the process' entry in PRT.* In the PRT entry the Swapper finds the Real Name

*Section 2.6 describes the Process Table (PRT).

of the process' Context Block. It brings this page into core. In the Context Block (CB) is a table, called the Active Page Table (APT), which contains pointers into the Process Memory Table (PMT). Entries in APT are marked as to whether the pages they point to are to be swapped in or not. The set of pages which are marked to be swapped in is called the Core Working Set (CWS) of the process. The Swapper scans APT and reads all CWS pages into core. When these reads are completed the process is said to be loaded and is available to be run on a CPU.

Figure 2.5 gives the format of an entry in APT. We now explain the various fields shown in the figure.

Use History: This field is used by the system to keep a history of references the process directs to the page the entry points to. It is updated periodically from the RF flag in PMT and used by the routines which maintain the Core Working Set.

Page Lock: It is possible to lock pages into core, that is, to exempt them from the algorithms which cause DIRTY pages to be written back on the drum and pages not in any Core Working Set to be released from core. The operating system can lock pages directly by turning on bits in the pages' entries in the Core Hash Table. Certain privileged User Programs will also need to insure that pages are kept in core. The Monitor provides a MCALL which can be used to do this. When a process executes this MCALL, the PAGE LOCK field of its CWS entry for the page will be set to a code identifying the lock bit in CHT for which the process is responsible.

Keep:
Lock: These fields are intended to allow a program to designate elements of its Core Working Set as more important than others. No operations on them are implemented in the current version of the Process Memory System, however.

DWS: In addition to the Core Memory Set there is another subset of APT called the Drum Working Set (DWS). It is the set of pages which are being kept on the drum for the process. It is a superset of the Core Working Set and is maintained entirely by the software parts of the operating system. The DWS bit in an APT entry is set if the page pointed to is in the process' Drum Working Set.

CWS: This is the bit the Swapper uses to determine whether an APT entry points to a page to be swapped in. It is set if the page is to be swapped in (i.e., is in the process' Core Working Set) and reset if it isn't.

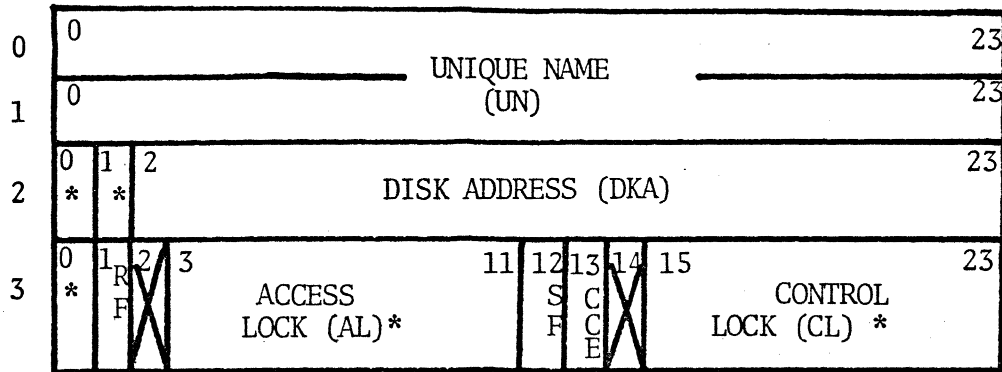
PMT Index: This is an index into the Process Memory Table and points to a Real Name of a page.

2.6 The Process Table (PRT)

A Process Table entry is kept for each active process in the system. The first three words of this entry give the real name (unique name + disk address) of the context block (CB) for the process. As described in section 2.5, the context block contains various tables: The PMT, a table of pages known to the process and the APT, a set of indices into PMT of currently active pages.

Thus if the swapper is given a process table index, it can (and does) determine which pages to swap.

Format of an Entry in a Process Memory Table



RDE - Read Error

RF - Reference Flag

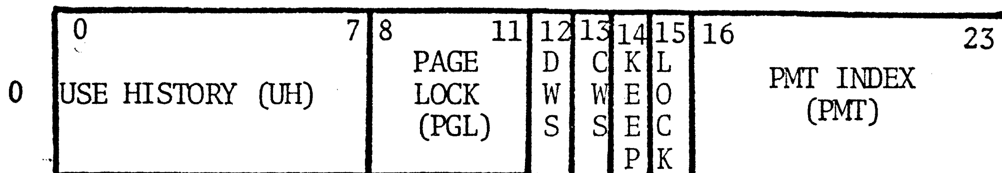
SF - Scheduled Flag

CCE - Class Code Error

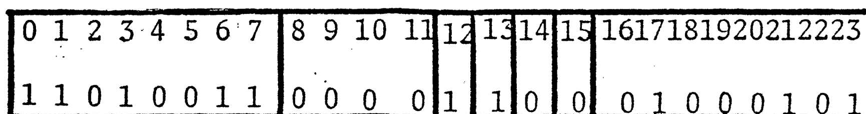
Figure 2.4

* Not used by the Memory Manager

Format of an Entry in an Active Page Table



An Example of an APT Entry



The entry tells us that the page whose Real Name appears in PMT[69] is in the process' Drum and Core Working Sets. The page is not locked in core for this process. Nor is it KEPT or LOCKed in the working sets. The process has made references to the page during

the last interval,

the next to the last interval,

the 3rd from the last interval,

the 6th from the last interval,

the 7th from the last interval,

Figure 2.5

Figures 2.6 and 2.7 detail the process table. Of particular interest to memory management are:

1) PRUN1

   PRUN2

   PRDK

   which give the real name of the context block for the process

2) PRSE

   which is an error word that the memory manager returns to indicate some problem which prevents it from swapping the process

3) In the Process Status Word (PRST),

   SWQ: a request for swapping-in has been put on the memory manager queue. When the memory manager considers it, this bit is reset.

   CBC: creating a request to read in a context block accompanies the setting of this bit (original request is a swapping-in queue request).

   PQ: after the context block has been read in, this bit accompanies the reading in of the individual pages of the process CWS. The CBC bit is turned off.

   LDD: Last leg of swapping in, this bit is set when all the pages have been read into core; PQ bit is turned off.

FORMAT OF THE PROCESS TABLE
(PRT)

Word	Contents
PRUN1	UNIQUE NAME 1
PRUN2	UNIQUE NAME 2
PRDK	ACT | X | DISK ADDRESS
PRPIW	| CO | ES | QT | AMC | CHI | | RAP | RSI | RT
PRSE	SWAPPER ERROR WORD
PRSW	LAST READ TIME (LRT) | DISK TRANSFER COUNT (KTC) | DRUM TRANSFER COUNT (DTC)
PRRTP	APT OVERFLOW | RTI POINTER (PRT POINTER)
PRRT	X | TIME OF NEXT REAL TIME INTERRUPT
PRST	MCT | PRI | | BLK (12) | WAQ (13) | CBC (14) | PDK (15) | PQ (16) | SWQ (17) | SCQ (18) | MSQ (19) | LDD (20) | RES (21) | RUN (22) | CPU (23)
PRCO	NIN* | PRD* | SCHEDULER FIELDS
	SCHEDULER FIELDS
PRPTR	| QUEUE POINTER (SCHEDULER, μ-SCHEDULER, ETC.)

*Bits only looked at by software

Figure 2.6

DESCRIPTION OF PRT BITS

CO: Carrier Off Interrupt

ES: Escape Interrupt

QT: Quit Interrupt

CHI: CHIO Interrupt

AMC: AMC Interrupt

RAP: Reduce Active Page Set

RSI: Run Scheduler Initially

RT: Real Time Interrupt

MCT: Millisecond Compute Time

PRI: Micro Scheduler Priority

BLK: Blocked

WAQ: Wake-up Queue

CBC: Context Block Considered

PDK: Process Delayed for Disk Transfers (No longer used)

PQ: Process Queued on Sector Read List

SWQ: On Swapper Request Queue

SCQ: On Scheduler Queue

MSQ: On Micro Scheduler Queue

LDD: Process Loaded

RUN: Process Running

CPU: CPU Number

RES: Resident Process

NIN: Non-Interactive

PRD: Process to be Destroyed

ACT: Active Process in PRT

Figure 2.7

3 INTRODUCTION - THE HARDWARE

This section describes the characteristics of the hardware used by the memory manager. We shall discuss first the processor, then the drum and disk, and finally the controllers. Figure 3.1 shows the logical relation between the various components.

3.1 The Processor

The processor that runs the memory manager is a microprocessor designed to provide a flexible and wide-ranging ability to perform varied processing and control functions in the system. The BCC 500 uses five of these processors, one of which is fully dedicated to memory management. The memory manager is also known as the Auxiliary Memory Control (AMC), processor.

This processor executes 90-bit microcode instructions, allowing a large number of possible functions to be carried out. Its cycle time is 100 nanoseconds. Basic boolean, shifting, and arithmetic functions are provided by the microprocessor. In addition to ten accumulator-like registers in the processor's arithmetic unit there are 64 scratchpad registers which allow quick access to variables which are referenced frequently.

Most of the memory management functions are implemented directly in the management processor's microcode, resulting in immense processing power for this (dedicated) purpose. Each of the other system microprocessors is similarly dedicated to one or more major system functions (including two that are "dedicated" to general calculation i.e., the CPU's). Except for the microcode and some easily wired special functions and conditions, each microprocessor is identical in construction. (The two CPU's are slightly different.)

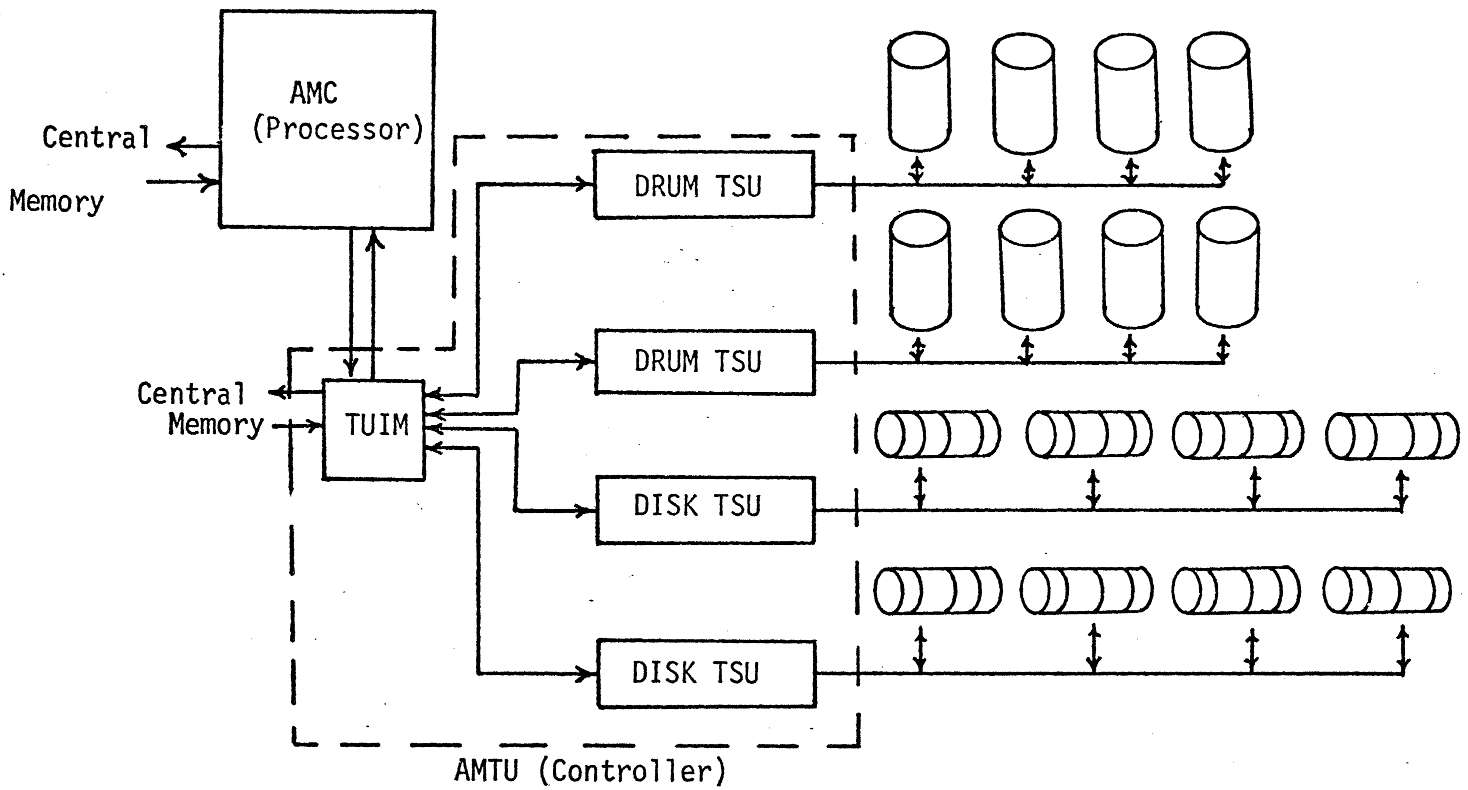
3.2 Drum and Disk Hardware

This section is concerned with the characteristics of the auxiliary memory hardware used by the memory manager. It is intended to describe everything about this hardware which can be observed from the vantage of the AMC.

3.2.1 Data and Address Formats

The AMC is connected to a block transfer unit, or controller called an auxiliary memory transfer unit (AMTU). Figure 3.1 shows the AMC, the AMTU, and their connections with the Central Memory. The AMC uses its own path to Central Memory to access system tables and APU instructions (elaborated on in Sections 5 and 7). The AMTU's path to Central Memory is used only for block transfers of data between Central Memory and the rotating devices. The path between the AMC and the AMTU is used for control: AMTU commands coming from the AMC and status information returning to the AMC.

The AMTU consists of four logically similar block transfer devices called Transfer Sub-Units (TSUs) and a Transfer Unit Interface Multiplexer (TUIM). Each TSU is capable of accepting a single command from the AMC to perform a specified block transfer, to monitor the many types of errors which may occur during a transfer, to report on the status of a transfer after completion, and to report certain other status information--especially rotational position information for the devices--required by the AMC. The TSUs are independent of each other and may perform transfers simultaneously. They share a single Central Memory port; and the TUIM serves



1 TUIM Transfer Unit Interface Mulitplexer

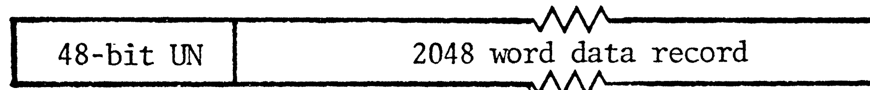
≤ 4 TSU's Transfer Sub-Units. ≤ 2 for drums, ≤ 2 for disks.

≤ 4 Units, drum or disk, per TSU

≤ 4 simultaneous data transfers, 1 per TSU

≤ 8 simultaneous seeks, 1 per disk

Figure 3.1: Auxiliary Memory System Configuration



Drum Page Format

Disk Page Format

Fig. 3.2 Drum and Disk Page Formats

to multiplex their (independent) requests to the Central Memory, resolving conflicting requests within it. The TUIM also serves as a routing and gathering point for control and status information passed from and to the AMC.

There are two types of rotating memory: drum and disk. The terms drum and disk refer primarily to fixed-head and moveable-head devices respectively, rather than to actual details of construction. (In the present hardware, drums are drums and disks are disks, however.) Drums have a higher transfer rate and no track selection latency (seek time).

The TSUs are consequently of two types: drum and disk. Each can have four devices or <u>units</u> connected to it. Each unit is a logically independent device which rotates and positions its arms independently of the other units (that is, if it has arms). A TSU can perform data transfers for only one of its attached units at a time.

Data is recorded on both drum and disk in units of 2048, 24-bit-word records called <u>pages</u>. Recorded with each page is a 48-bit unique <u>name</u> (UN). Also associated with each disk page is a <u>header</u> which contains the physical address of the page. Figure 3.2 illustrates this. Note that on the disk, header, UN and data are three separate records; while on the drum, UN and data form a single record. Each data record has a checksum. A drum checksum is of length 24 bits and a disk checksum is of length 48 bits. Finally, each word is recorded on disk with an additional parity bit.

The drums were designed specifically to transfer at a high rate. The disk is not required to be quite so fast as the drum, as its main purpose in the system is bulk storage. When processes become active or files are

accessed their pages primarily reside on the drum; they move into core to be accessed and then back out onto the drum. When they are no longer actively being accessed they return to the disk.

Along with the discussion of the drum and disk, various terms that relate to them will be defined.

Drum: A page on a drum is defined by its sector and band position. A sector is an angular segment of the drum required to hold one page. As the drum rotates, succeeding sectors come under the heads. There are 24 sectors or pages around the drum. Information is transferred to/from drum in 24 bit-parallel fashion, i.e., 24 heads are used simultaneously. Since the drum is equipped with approximately 1100 heads, it is necessary to specify which grouping of 24 heads to use during a given transfer. This grouping is called a band. There are 42 bands on a drum, allowing for storage of 42 X 24, or 1008 pages (i.e., a total of 1008 X 2048 = 2,064,384 words) per drum.

With a rotation period of 33.3 msec, the drum transfers information at the rate of 750 pages/sec ($1.5 \cdot 10^6$ words/sec). This is a rather high rate; however one must recall that this is a potential rate and not necessarily an actual one. It is the responsibility of the memory manager to make the most of this potential rate.

Disk: The disk is even more unconventional than the drum. It is a large file with 13 disks each 4½ feet in diameter. Like the drum, it transfers using a group of heads in parallel to define a logical "band". Since only six heads are used for the 24-bit words being transferred, however, the transfer is partly serial. The heads of a given band are located on a

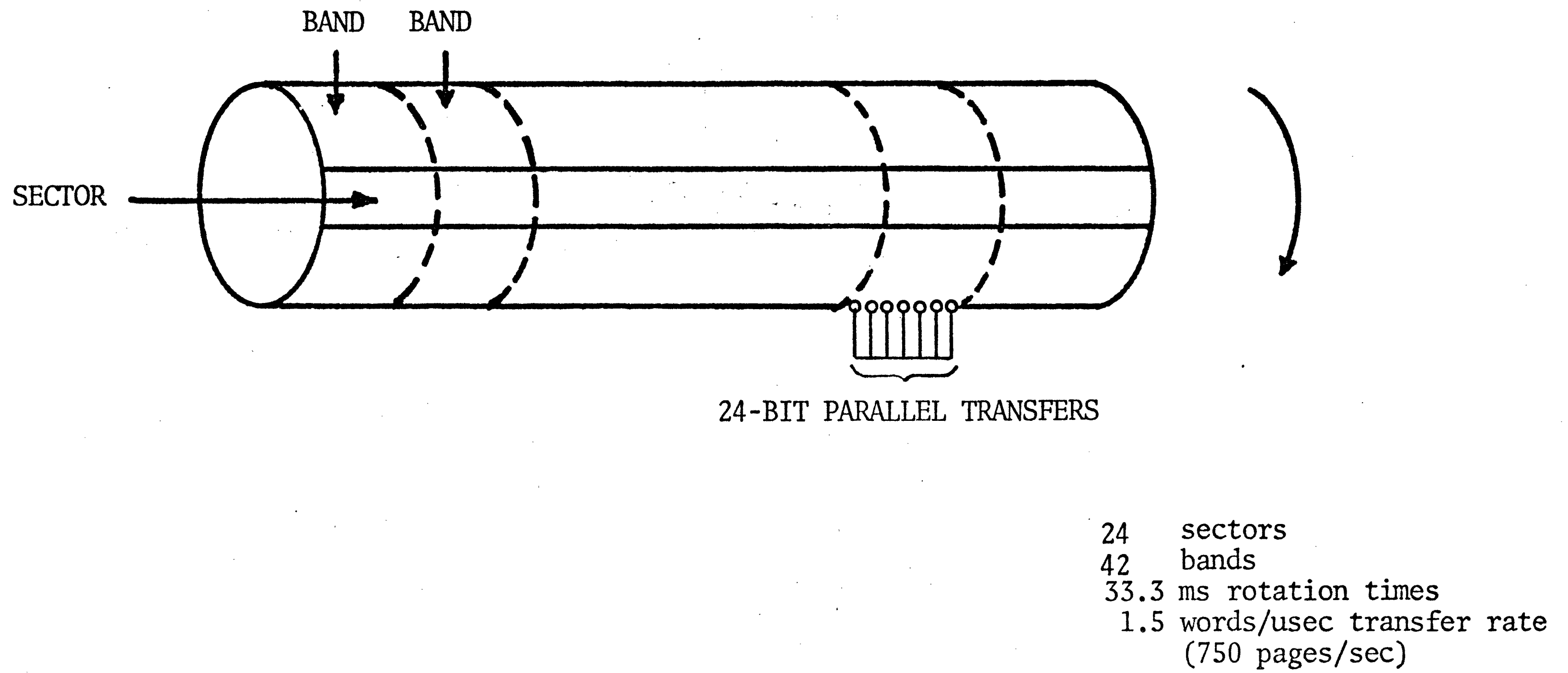
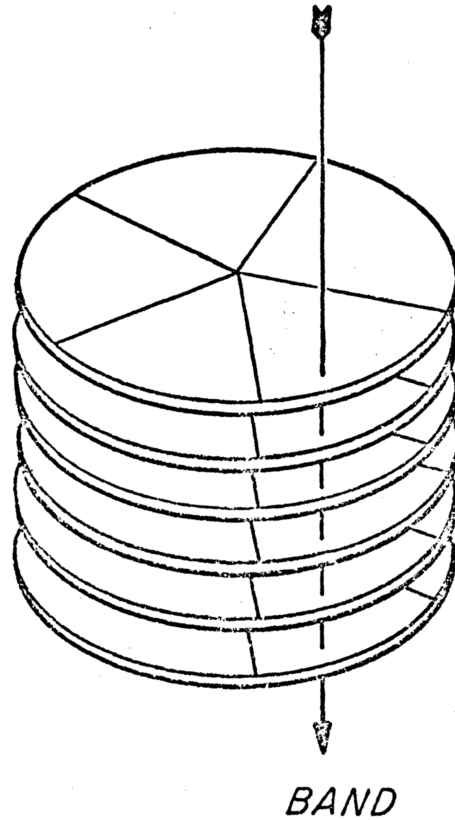


Fig. 3.3a Drum Characteristics

DISK



5 SECTORS

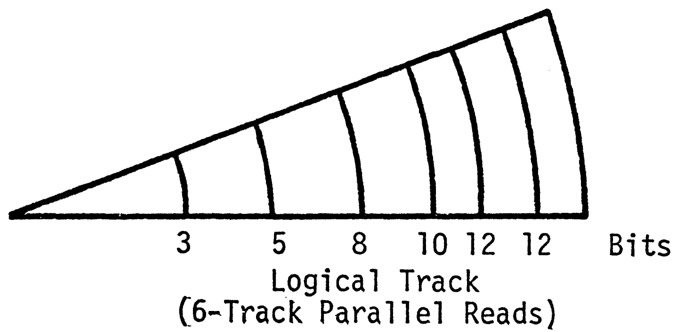
24 BANDS

256 CYLINDERS

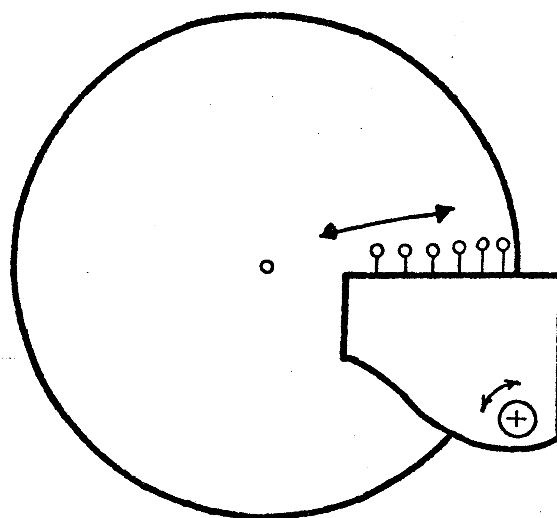
50 msec rotation

.27 words/usec (135 pages/sec)

Figure 3.3b



Logical Track
(6-Track Parallel Reads)



Radial Head Movement, Cylinder Positioning
(256 Possible Positions)

Figure 3.3c

radius of one side of one of the disks. On a disk the head nearest the outer edge moves over a greater circumference than the inside head. (The disk is depicted in figures 3.3b and 3.3c) Thus more bits can be stored under the outside head than the inside head, and the bit rate per head is thus a function of the head's radius. Figure 3.3c shows the relative number of bits transferred during a unit of time. Note that these numbers add up to 50 bits, 2 for parity and 48 for two words. Most disks read words sequentially off tracks, as well as use worst case (low) densities that are the same no matter where the head positions is. The 500 method allows more density and greater transfer rate.

Whereas on the drum the heads are fixed, on the disk they are moveable. On the 500 disk there is an angular positioning arm which moves the heads over the correct logical track, or cylinder.

There are 256 angular positions. For components of the disk address, we use the terms band, sector, and cylinder. A sector is that angular portion of the disk necessary to store one page. There are five sectors per revolution on the disk. Figure 3.3b depicts the organization of the disk. The disk has 13 platters with 26 surfaces. Each surface corresponds to a band. A cylinder is one of the 256 angular positions. (The term cylinder - reminiscent of a drum - derives from the fact that for each track positioning the disk addressing structure otherwise resembles a drum.) In all, there are 30,720 pages (or 63 x 10^6 words) on a disk unit. The disk has a 50 ms rotation time, with an average 4.8 μsec. to transfer a (double) word

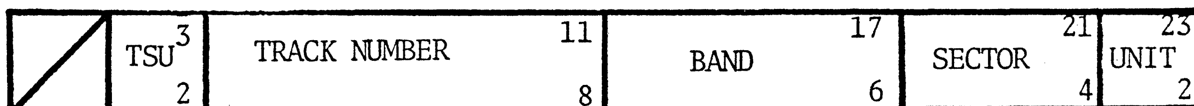
All BCC 500 rotating memory parameters are summarized in Figure 3.4.

Drum:	24	sectors/revolution
	42	bands
	1,008	pages/unit or 2,064,384 words/unit
	2	units (Hawaii configuration)
	33.3	ms rotation time
	1.5	words/μsec transfer rate
	~30	μs sector gap
Disk:	5	sectors/revolution
	24	bands
	120	pages/cylinder or 240,000 words/cylinder
	256	cylinders/unit
	30,720	pages/unit or 62,914,560 words/unit
	2	units (Hawaii configuration)
	50	ms rotation time
	.21	words/μsec transfer rate
	~50	μs sector gap
	65-225	ms seek time. This subject is discussed in detail elsewhere

Figure 3.4 Parameters of BCC 500 Rotating Memories

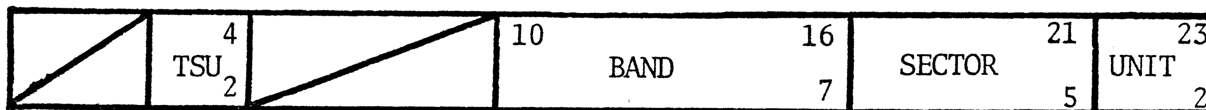
AMC Disk address internal form:

The disk unit has 5 sectors per band, 24 bands per cylinder, and 256 cylinders. There are two TSUs which may control disks (the Hawaii configuration has only one, however). Each TSU may have up to four disk units attached to it. This information has been packed into one word in order to keep tables small.



AMC Drum address internal form:

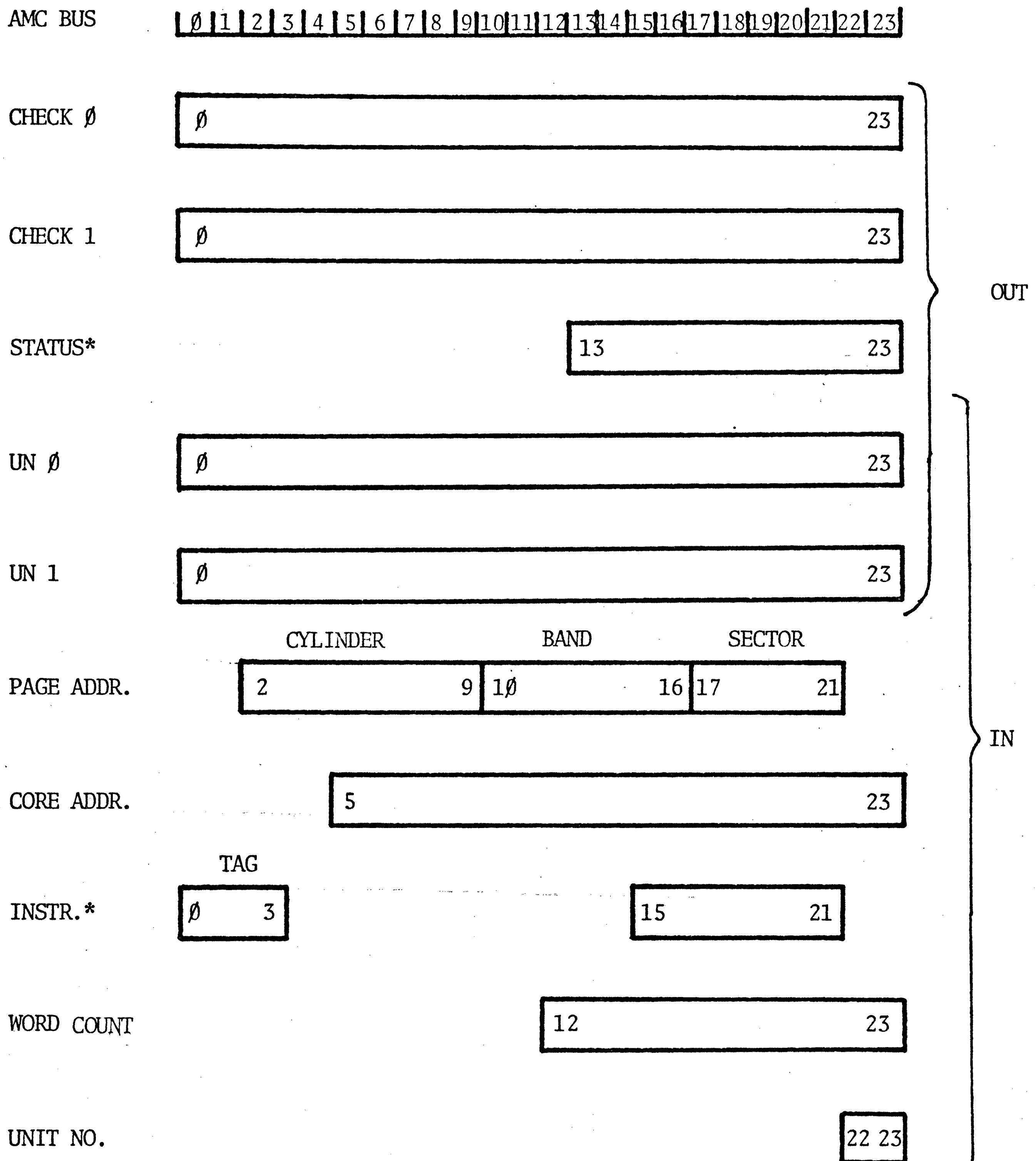
The two million word drum has 24 sectors per band and 42 bands. Since space in the address word is not as critical in the drum address, the fields have been adjusted to correspond to the device address fields in the TSU.



The address of a page, as accepted by the TSU, is contained in one word with the following format:

2 - 9	cylinder (disk only)
10 - 16	band
17 - 21	sector

The other bits in the word are not used. This format is shown in Fig. 3.5a.



*Amplified on next page

Fig. 3.5a TSU Control Registers

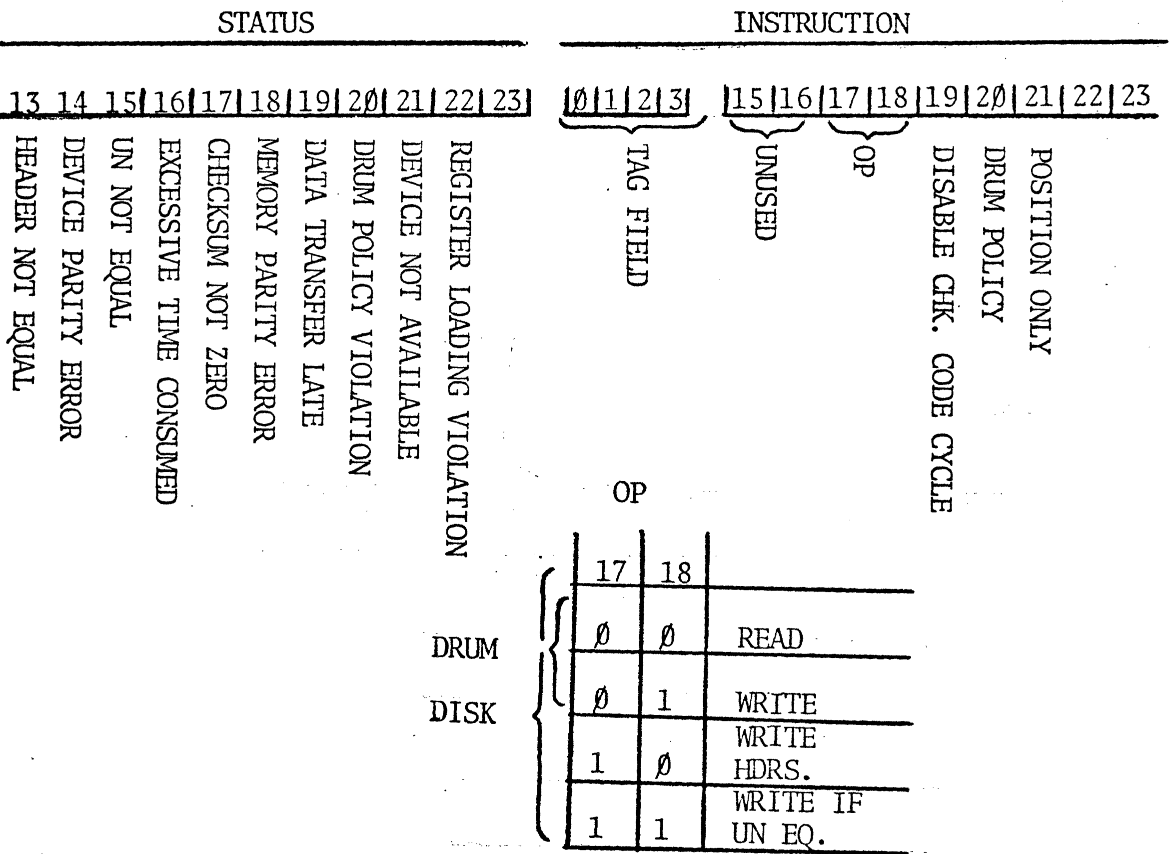


Figure 3.5b: TSU Registers

3.2.2 Position Counters

Each TSU has four registers called position counters (PCs), one for each unit. The format of a PC is shown in Fig. 3.6a. The four bits marked * show the same in all four PCs; they are identical copies of indicators and are discussed in Section 3.2.4. The rest of the PC gives information about the arm position and rotational position of its unit, as follows.

- the sector number tells which sector is currently under the heads
- for the disk, the cylinder number tells which cylinder is currently under the heads or being sought for. For the details of how to interpret this field and the related CYLINDER VERIFIED bit, see Section 3.2.10.
- the position within the sector (PWS) field tells approximately where in the sector the heads currently are.

Figure 3.6b tells how to interpret the PWS field for the drum. The reasons for the strange intervals are historic.

Note especially the definition of the end of sector (EOS) time as the time the PWS field goes from 15 to Ø. This is the reference point from which most activity in the TSU is measured.

If the POSITION NOT VALID bit is on, the information in the PC is meaningless because it is being incremented. If this bit is on, the PC is merely reread until its bit goes off.

The PC for a disk unit is not affected in any way by a seek taking place for that unit.

3.2.3 TSU Control Registers

A TSU (each TSU operates completely independently of the others) has two sets of control registers. One set, called the functional registers (FR), contains the instruction currently being executed. The other set, called the holding registers (HR), can be read or written by the AMC. Whenever an instruction is completed, and at certain other times, the FR are exchanged or swapped with the HR. Precise details of this process are presented below. Figure 3.5a summarizes the register formats and should be referred to while reading this section.

Timing in the TSU is geared to the transfer of pages between a rotating device and central memory. The time at which most things happen is the EOS time (see Section 3.2.2), the time at which all the data of a sector, and any auxiliary information (checksums, etc.) which the hardware may suffix to the data, has passed by the heads. At this time the inter-sector gap is under the heads; after it has passed by, information related to the next sector begins to come under the heads. We will call this time T, sometimes with a subscript to denote the sector which has just passed by. An EOS time occurs in each unit at regular intervals. The unit addressed by the functional UNIT NO (the selected unit) provides the EOS time for the TSU. A TSU EOS occurs at regular intervals except when a new unit is selected; this case is discussed below.

The TSU works in the following way. The AMC loads the control registers in the order: instruction register, core address, word count, page address, UN, and unit number, into the HR (FR are not generally accessible to the AMC).

POSITION COUNTER (3-11 UNUSED IN DRUM TSU)

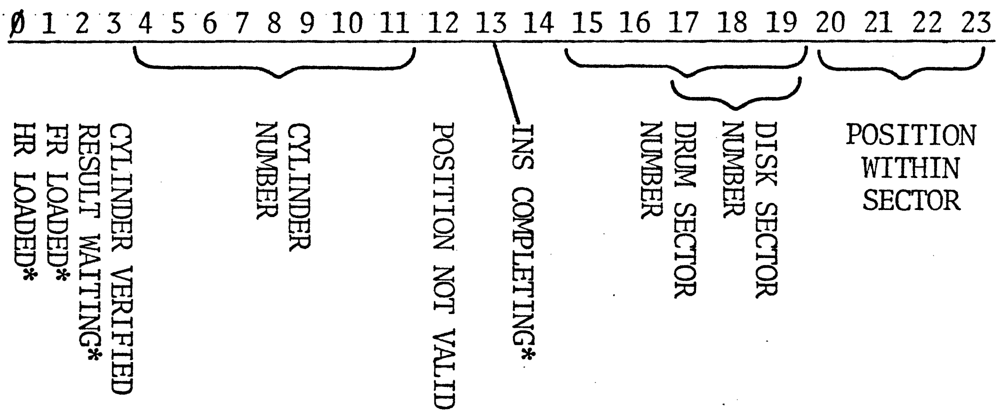


Fig. 3.6a Position Counter (PC) Format

Ticks	PWS	Length of Interval	What is Happening
∅ - 23	∅	24	Inter-sector gap
24 - 70	1	47	Preamble, data starts
71 - 134	2	64	
135 - 157	3	23	
158 - 165	4	8	
166 - 212	5	47	
213 - 276	6	64	Data
277 - 299	7	23	
300 - 307	8	8	
308 - 354	9	47	
355 - 418	10	64	
419 - 441	11	23	
442 - 449	12	8	
450 - 496	13	47	
497 - 560	14	64	Data, checksum, post-amble, dead time
561 - 584	15	23	Dead time

One tick = 2.4 μs.   584 ticks = 1.4 ms = 1 sector time

Transition from PWS=15 to PWS=∅ is End-Of-Sector (EOS)

Fig. 3.6b Drum Record Timing

We assume sector i is now under the heads. When the EOS time T_i occurs, the TSU swaps the registers. This takes about 1 μsec. The TSU then starts "executing" the instruction in the FR i.e., it attempts the indicated transfer or arm positioning. Let us say the indicated transfer is for sector j. Then the TSU waits for sector j to come under the heads and does the transfer. At T_j the registers are swapped again, and the AMC can then look at the HR to find out what happened during the instruction. At the same time, if the HR were loaded again after T_i but before T_j, the new instruction is being executed by the TSU. The AMC thus has one sector time to examine the results of the transfer done in the previous sector and set up an instruction for execution in the next sector and can thus accomplish continuous transfer.

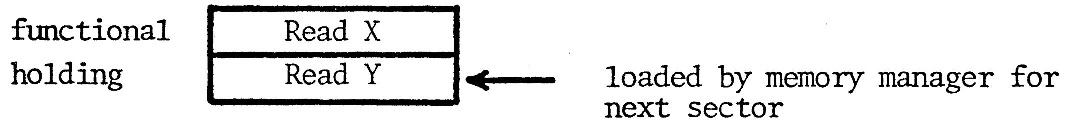
Matters are slightly complicated if the unit is changed since the units are not rotationally synchronized. The swap occurs as usual at T_i. Then the TSU starts to look for an EOS on the newly selected unit. This EOS is called a synchronizing EOS, and until it occurs the TSU is suspended. The synchronizing EOS serves only to activate the suspended TSU. When it occurs, execution of the FR instruction begins, exactly as it would have at T_i if there had not been a unit change. The second EOS on the new unit is the next effective EOS after T_i. It follows that the interval between TSU EOS times will be a variable and as long as twice the normal interval following a unit change.

There is a special feature in the hardware to take care of a dual-positioner disk, in which the two units rotate synchronously. When the TSU switches from one unit of the dual positioner to the other it behaves as though

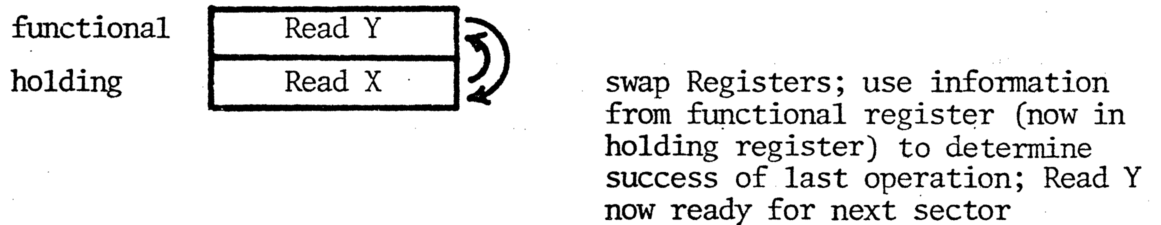
(Instruction register only)

1) during sector --

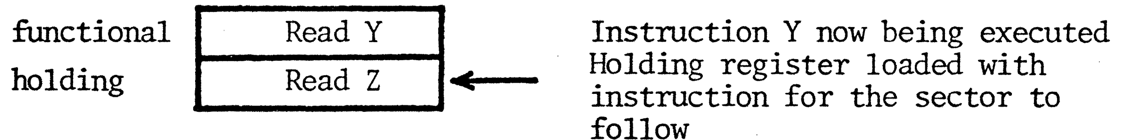
2) shortly before sector ends --



3) end of sector (EOS) --



4) before end of succeeding sector --



The above pattern is repeated with the functional register instruction being executed while the holding register is loaded in preparation for the next sector. Then between sectors, the registers are again swapped.

Fig. 3.7 TSU Register Loading

no unit change had occurred, i.e. it never becomes suspended and no synchronizing EOS is required. If this case were not handled specially, a unit switch on the dual positioner would always cause the TSU to be suspended for a full sector-time.

When the AMC loads the registers, it must load the instruction register first and the unit number last. If the instruction register is not loaded between T_i and T_{i+1}, no instruction will be executed during sector i+2. In this case the TSU is said to be idle. A swap occurs as usual at T_{i+1} to bring the results of the instruction which was loaded at T_i back to the HR for examination. If the HR are not loaded during sector i+2, however, no swap occurs at T_{i+2}. In this case the TSU remains idle during sector i+3, and continues to be idle until an instruction is loaded into HR. If this happens during sector j, a swap will occur at T_j and the TSU will cease to be idle at that point.

If a swap occurs after the instruction register has been loaded but before the unit number has been loaded, a register loading violation occurs. The instruction is taken as a NOP and the status register will indicate the violation after execution of the NOP is complete. See Section 3.2.5 for details of instruction timing, and Section 3.2.9 for how the loads are handled.

3.2.4 Indicators

The four indicator bits marked * in any position counter are each copies of four indicators (see Section 3.2.2 for the significance of the rest of the word) which record the progress of an instruction through the cycle just described. When the holding instruction register is loaded, the HR LOADED

bit (bit Ø of any position counter) is set. HR LOADED is cleared when the registers are swapped, so it is set exactly when there is an instruction loaded (or partially loaded) in the HR and awaiting execution.

The FR LOADED bit (bit 1 in the PCs) is set from HR LOADED when the registers are swapped. It is therefore set exactly when there is an instruction in the FR being executed or awaiting execution. Note that this bit comes on when an instruction enters the FR even if the sector addressed by the instruction is not under the heads. It therefore provides no information about whether the data transfer is actually underway.

This information is provided by the INS COMPLETING bit (bit 13 in the PCs) which is on if FR LOADED is on and the instruction will complete at the end of the current sector on the selected unit. More precisely FR LOADED may be on and INS COMPLETING off only if

1) the TSU is suspended, or

2) the instruction has DRUM POLICY=Ø, and POSITION ONLY=Ø and address sector i is not now under the heads (or, in the case of a disk write, CYLINDER VERIFIED is off).

In all other cases FR LOADED implies INS COMPLETING. These cases may be classified as follows:

3) the instruction has DRUM POLICY=1

4) the instruction has POSITION ONLY=1

5) the instruction has DRUM POLICY=Ø, POSITION ONLY=Ø, addresses sector i and sector i is now under the heads (and, in the case of a disk write, CYLINDER VERIFIED is on).

6) there is a register loading violation

7) the instruction consumes excessive time (see Section 3.2.7).

The definition implies that INS COMPLETING is always on for exactly one sector per completed instruction.

Finally, the RESULT WAITING bit (bit 2 of the PCs) is set from FR LOADED when the registers are swapped. It is cleared when AMC reads the STATUS register. It thus indicates that there is information waiting to be collected about the fate of the last command. The TSU will not swap new values from the FR into the HR if the RESULT WAITING bit is on. This situation occurs if the AMC has not yet read the results of a previous transfer which would be clobbered by the results of the current transfer.

The three bits HR LOADED, FR LOADED and RESULT WAITING may be thought of as resting places for a single bit indicating the presence of an instruction which goes from AMC to HR to FR to HR and back to AMC.

All four indicator bits may change state about 1 μsec after a TSU EOS. The INS COMPLETING bit may be set about 1 μsec after a synchronizing EOS. These are the only times that the indicators can change state.

3.2.5 TSU Instructions

There are two instructions which may be given to a drum TSU. All drum instructions have POSITION ONLY=Ø. Normally drum instructions have DRUM POLICY=1 and DISABLE CHECK CYCLE=Ø (see below).

OP=Ø: Transfer WORD CNT words, from the beginning of drum page PAGE ADDR to central memory locations CORE ADDR through CORE ADDR+WORD CNT-1. At the end

of the read the UN of the page is left in UNØ and UN1. A checksum for the page (<u>not</u> including the UN) is in CHECKØ; if it is non-zero the data has been read from the drum incorrectly, i.e., there has been a read error. In this case the CHECKSUM NOT ZERO bit in STATUS will be set. PAGE ADDR will be pointing to the next page, CORE ADDR to the first unreferenced core location and WORD CNT will be 77757776B (i.e., -2 in its context); these three registers are normally of little interest after the read. A variety of useful information discussed in detail in Section 3.2.7 is left in STATUS. On a read the checksum is computed for the entire page, regardless of the WORD CNT, as long as it is >Ø.

OP=1: Transfer the UN in UNØ and UN1 and the contents of central memory locations CORE ADDR through CORE ADDR+WORD CNT-1 followed by 2048-WORD CNT zero words to the drum page at PAGE ADDR. CHECKØ and CHECK1 are of no value after this instruction. STATUS reports unusual conditions as for a read, and the addresses and count are left as for a read.

There are five instructions which may be given to a disk TSU. Four are data transfer instructions and have POSITION ONLY=Ø. These instructions may have DRUM POLICY on or off (see Section 3.2.6). They normally have DISABLE CHECK CYCLE=Ø.

OP=Ø: Read, which works exactly like drum read, except that the disk has two checksum words, CHECKØ and CHECK1. A disk read is not affected by CYLINDER VERIFIED, but it will set CYLINDER VERIFIED if there is no error; see Section 3.2.10.

OP=1: Write, which works exactly like drum write, except that it is not executed until CYLINDER VERIFIED for the selected unit is set; see Section 3.2.10.

OP=2: Write header, which is an instruction used only by hardware maintenance personnel and is used only at disk maintenance times.

OP=3: Write with UN check. This instruction is like write except that instead of writing UNØ and UN1 it compares them with the values recorded on the disk. If the disk values agree the write proceeds normally. If not the write is aborted and the UN NOT EQUAL bit in STATUS is set. This is the only condition which sets this bit.

The fifth disk TSU instruction has POSITION ONLY=1. The value of OP is ignored in this case. This instruction causes the disk to move the arms to the cylinder specified by PAGE ADDR. The other registers are ignored. No data is transferred. This instruction requires one sector time for execution plus any time during which the TSU is effectively suspended due to lack of cylinder verification. DRUM POLICY must be Ø for this instruction.

The DISABLE CHK CODE CYCLE bit alters the method of computing the checksum words. Normally for the drum (disk) CHECKØ (CHECKØ and CHECK1) is computed by starting with the first word (double word) and then as each word (double-word) arrives doing an Exclusive OR of it with CHECKØ (CHECKØ and CHECK1) and cycling the result left 1. The cycle is suppressed by this bit. The suppression of the check code cycle diminishes the effectiveness of the check code and is intended for hardware check-out purposes only. All other use of drum or disk involving data must specify cycling of the check code.

The TAG FIELD is not used or modified in any way by the TSU. It is

provided solely for the convenience of the AMC, which by using it may attach a 4-bit tag to an instruction. The tag bits are copied, along with the rest of the instruction, from HR to FR and back to HR. The tag is intended to help identify the loss of instructions due to register loading violations.

3.2.6 Instruction Timing

If DRUM POLICY=1 in any instruction and the unit is not changed, the sector position of the page addressed must be the one which will be under the heads immediately after the swap which brings the instruction into the FR, the arm position must be the current arm position (only relevant for disk), and CYLINDER VERIFIED must be set for a disk write. If this is not the case, the instruction is aborted and DRUM POLICY VIOLATION is set in STATUS. If the unit is not changed the instruction requires exactly one sector time to execute.

If DRUM POLICY=1 in any instruction and the unit is changed, the page addressed must be the one which will be under the heads after the synchronizing EOS swap. In this case the instruction requires somewhere between one and two sector times to execute.

If DRUM POLICY=Ø, this restriction is not enforced. In this case a transfer instruction may require as much as one full device revolution, plus seek time if the arm position for the page addressed differs from the current arm position and the instruction is a write. The instruction will complete at the first EOS time for the sector addressed after the arms have verified a new position, if required. A POSITION ONLY instruction requires exactly one sector time to

execute regardless of the PAGE ADDR, unless units are switched, in which case it requires between one and two sector times.

A swap can occur only at a TSU EOS. A swap will occur at each TSU EOS where

HR LOADED or (INS COMPLETING and not RESULT WAITING)

3.2.7 Status Register

The STATUS register contains an assortment of bits which report on errors which may occur during execution of an instruction. They are catalogued here, together with an exhaustive description of the conditions which cause them to be set. The FR STATUS is cleared when a swap occurs. It may then accumulate bits until the next swap, at which time its contents are copied into HR STATUS and it is cleared again. A successful instruction will have a cleared STATUS register.

In the absence of any other comment, an instruction which causes a STATUS bit to be set does nothing and requires exactly one sector time (plus unit change time). Such an instruction leaves WORD CNT = 77757777B and CORE ADDR and PAGE ADDR unchanged. When a data transfer has started it can be aborted by DATA TRANSFER LATE; memory parity errors do not abort the transfer.

1) REGISTER LOADING VIOLATION is set if the instruction register has been loaded when a swap occurs but the unit number has not. It appears in FR STATUS after the swap which interrupts the loading, and then in HR STATUS after the next swap. See Section 3.2.9 for a discussion of register loading. RLV will also occur if the AMC attempts to

load the HR when they contain the results of a previous transfer (RESULT WAITING=1). This will happen if, for example, a swap occurs before the AMC can even attempt to load the instruction register. The RLV will appear in the HR at the end of the interval in which this happens, or as soon as the RESULT WAITING bit is cleared (STATUS is read).

2) DEVICE NOT AVAILABLE is set if the unit addressed by the UNIT NO register is not connected to the TSU, has power turned off, or is in some bad state.

3) DRUM POLICY VIOLATION is set if DRUM POLICY=1 and the page addressed by PAGE ADDR is not under the heads when the instruction arrives in the FR. (See Section 3.2.6.)

4) DATA TRANSFER LATE is set if the memory fails to deliver a word soon enough (write) or to absorb one soon enough (read). When it is set the rest of the transfer is aborted; in a write zeros are written. Timing is not changed.

5) MEMORY PARITY ERROR is self-explanatory. Timing is not changed and the transfer is not aborted.

6) CHECKSUM NOT ZERO is set at the end of a read if CHECKØ (drum) or CHECKØ and CHECK1 (disk) are non-zero, indicating that an error has occurred. Timing is not changed. Note that the UN is <u>not</u> included in the checksum.

7) EXCESSIVE TIME CONSUMED is set if the instruction remains in the FR for more than 1 second. This can happen if the page address is for a non-existent page or if the seek mechanism on the disk fails. The instruction does nothing. INS COMPLETING is set at the next TSU EOS after the 1 second has elapsed, and the instruction completes at the following TSU EOS as usual. ETC can only happen on an instruction with DRUM POLICY=POSITION ONLY=∅ which has a PAGE ADDR whose sector number is too large for the unit (or as a result of hardware failure).

8) UN NOT EQUAL (disk write with UN check only) is set if the UN recorded on disk differs from UN∅ and UN1. No data is transferred, the disk is not changed, and timing is not changed.

9) DEVICE PARITY ERROR (disk read or disk write with UN check only) is set if a word is read from the disk and has the wrong parity. Timing is not changed. One parity bit is recorded with each word on disk, in addition to the two word checksum at the end of the record. This error can occur when reading the UN as well as when reading data. The cases cannot be distinguished.

10) HEADER NOT EQUAL (disk transfer only) is set if the page address recorded in the header for the page disagrees with the address in the FR PAGE ADDR register. This can happen only if the header is wrong or was read incorrectly.

The following bits abort transfers before central memory or drum/disk is changed: REGISTER LOADING VIOLATION, DEVICE NOT AVAILABLE, DRUM POLICY VIOLATION, EXCESSIVE TIME CONSUMED, UN NOT EQUAL, HEADER NOT EQUAL.

DATA TRANSFER LATE aborts transfers after core or disk is changed.

The following bits do not affect transfers (but the data is probably bad): MEMORY PARITY ERROR, DEVICE PARITY ERROR.

The other STATUS bits do not affect the transfer.

Only the following bits can be set by a POSITION ONLY command: REGISTER LOADING VIOLATION, DEVICE NOT AVAILABLE, DRUM POLICY VIOLATION (always, unless DRUM POLICY=Ø).

3.2.8 Attentions

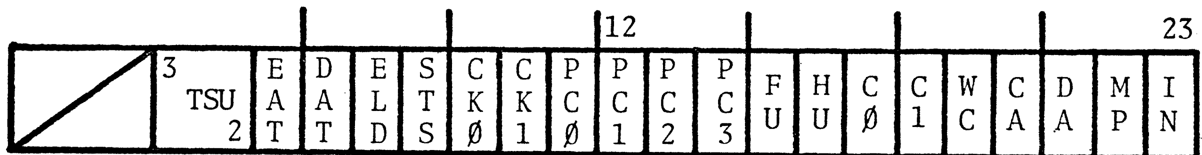
The TSU sends an attention to the AMC shortly after every TSU EOS or synchronizing EOS if attentions are enabled from that TSU (see Section 3.2.9). In other words, an attention is sent, if enabled, at every EOS on the selected unit. An attention will occur on the EOS before the addressed sector, and another one on the EOS after the addressed sector.

There is no way to tell which TSU sent an attention except to go and look at the various registers. It is of course possible for several TSUs to send attentions at the same time.

3.2.9 Select Register in the TSU

This register (Fig. 3.8) is loaded from one of the AMC registers (Z register) when an ALERT is executed (a microprocessor special function). There are four select registers, one for each TSU. Two bits in Z determine which TSU is addressed by the ALERT. Any bit may be set in the Select Register. The select register bits correspond to TSU registers which are to be read or written by the AMC, as performed to the E2 bus or from the Z register by PIN

or POT special functions. Data is sent to all registers for which a bit is set. The data transferred to the AMC from the TSU is the OR of all registers for which a bit is set in the select register.



where TSU = TSU Number

EAT = Enable attention

DAT = Disable attention

ELD = Enable load

Figure 3.8: TSU Register Loading (Select Register)

The remaining bits are select bits. If Enable load is set, the register will be loaded when a pot is executed. If Enable load is not set, the appropriate registers will appear on the E2 bus.

STS = Status register

CKØ = Check code Ø

CK1 = Check code 1

PCØ = Position Counter Ø

PC1 = Position Counter 1

PC2 = Position Counter 2

PC3 = Position Counter 3

FU = Functional Unit register (the only functional register that is addressable)

HU = Holding Unit register

CØ = Class Code (Unique Name) Ø

C1 = Class Code (Unique Name) 1

WC = Word Count

CA = Core address

DA = Device address

MP = Map address

IN = Instruction register

Registers can be loaded only in the following order:

1) instruction register
2) any other registers except unit number
3) unit number.

The rule is enforced in the following way: there is a flag OKL which indicates that it is OK to load registers. This flag is

1) set by loading the instruction register when HR LOADED is off
2) reset by loading the unit number, or by a swap.

If any attempt is made to load any register other than the instruction register and OKL is off, the load is aborted. A register loading violation occurs if OKL is set when a swap occurs. The most important consequence of all this is that if a swap occurs in the middle of register loading, the results of the old instruction, now in the HR, are not destroyed by subsequent loads.

3.2.10 Disk Seeks and Position Verification

Whenever a disk unit is given an instruction for which the arm position (cylinder) in PAGE ADDR differs from the arm position in the last instruction to that unit, it will move the arms to the newly specified position.

The timing of the arm movement operation ranges from about 65 msec to 220 msec depending on the arm displacement and on which of the eight positioning pistons (one corresponding to each bit of the cylinder number) change state.

An estimate of the time required for the operation can be computed using the expression below:

$$56 + 4 * \sqrt{|\text{new cylinder} - \text{old cylinder}|} + f(\text{new cylinder, old cylinder}) \text{ msec}$$

where f(x,y) is given by the following table:

Weight of most significant bit which x differs from y	Time
128	100
64	50
32	25
16	13
8	6
4	3
2	1
1	0

Each disk unit has a CYLINDER VERIFIED flag (CV) which is bit 3 of any of its PCs. Whenever a seek is initiated on unit u, CV_u is cleared. This is the only way to clear CV_u. It can be set under two circumstances:

1) A read is done from the unit without any errors.
2) The time since the seek was started exceeds AV ms (currently 300 ms). This time is called the automatic verification interval. This method of setting CV_u is independent of whether u is selected.

The rationale behind all this is as follows: when a seek is done, there is a period during which the arms move to a new position, and a subsequent period during which they oscillate around the new position. AV is chosen large enough to ensure that any seek will be completed in that much time. Faster position verification can be obtained by successfully reading a sector. Since a sector is 10 ms long and has 48 checksum bits as well as a parity bit on each word, it is possible to feel confident that the arms have settled down if no errors occur in the read of the sector. Note that any WORD CNT > Ø is sufficient to verify the position.

A drawback of this scheme is that it becomes necessary to distinguish between parity and checksum errors caused by bad data and those caused by incomplete positioning. Since data errors are expected to be rare, it is quite satisfactory to examine CV after the transfer. If it is set, the error is a data error. As a consequence, an attempt to read a bad page may take as much as AV+R ms, where R is the read time.

Needless to say, read instructions are not affected by CV. A write, however,

is not allowed to occur unless CV is on. The reasons for this are that

1) there is no parity or checksum error detection on a write
2) more important, if the arms are still moving a write may spread destruction across several cylinders of a disk surface.

A DRUM POLICY=1 write will cause a policy violation if CV is off; a DRUM POLICY=0 write will wait until CV is on before executing.

3.2.11 TSU State

When the Main Loop of the Memory Manager determines that a TSU needs servicing, it puts the state of the TSU into central memory. The routine also supplies the pointer to the cleanup buffer (described in data structures section) and the TSU number as part of the state. The APU (software) is concerned with this information so that the state is put into the local address space of the APU code.

56	Pointer to buffer (also in B register)
57	TSU #
6∅	Instruction register
61	Map register
62	Device address register
63	Central memory address register
64	Word Count register
65	Class code register #1
66	Class code register #∅
67	Holding Unit Register
7∅	Functional Unit Register

71	Position Counter for Unit Ø
72	Position Counter for Unit 1
73	Position Counter for Unit 2
74	Position Counter for Unit 3
75	Check word #1 register
76	Check word #Ø register
77	Status register

4. INTRODUCTION - DATA STRUCTURES

This section describes the main data structures used by the memory manager.

4.1 Queues

The fundamental data structure used in the organization of the AMC is the queue. A request will start off on a certain queue, and as it is partially serviced, be passed along to succeeding queues until its completion.

Thus a "general" request such as "swap in a process" will be broken down into requests for specific pages of the process to be read from the drum and disk. If one were to freeze the AMC processing at a specific point, by looking at the various queue entries, one could get a fairly comprehensive picture of the state of processing that the AMC was in at the time.

To illustrate the fundamental role of the queue in the memory manager, we shall follow the swapping in of a process beginning with a high level request to read in a process. The request starts on the

SWAP QUEUE

in the form of a request node. Figure 4.1 shows the format of a request node while Section 4.2 describes the request node in detail. This is the same basic format that the AMC uses for all its various request nodes. This request node has the unique name of the context block of this process. As shown earlier in the section describing context blocks, the context block contains the unique names of all pages belonging to the process. We start by reading in the context block. To get the context block in, we put the request on a

DRUM SECTOR QUEUE

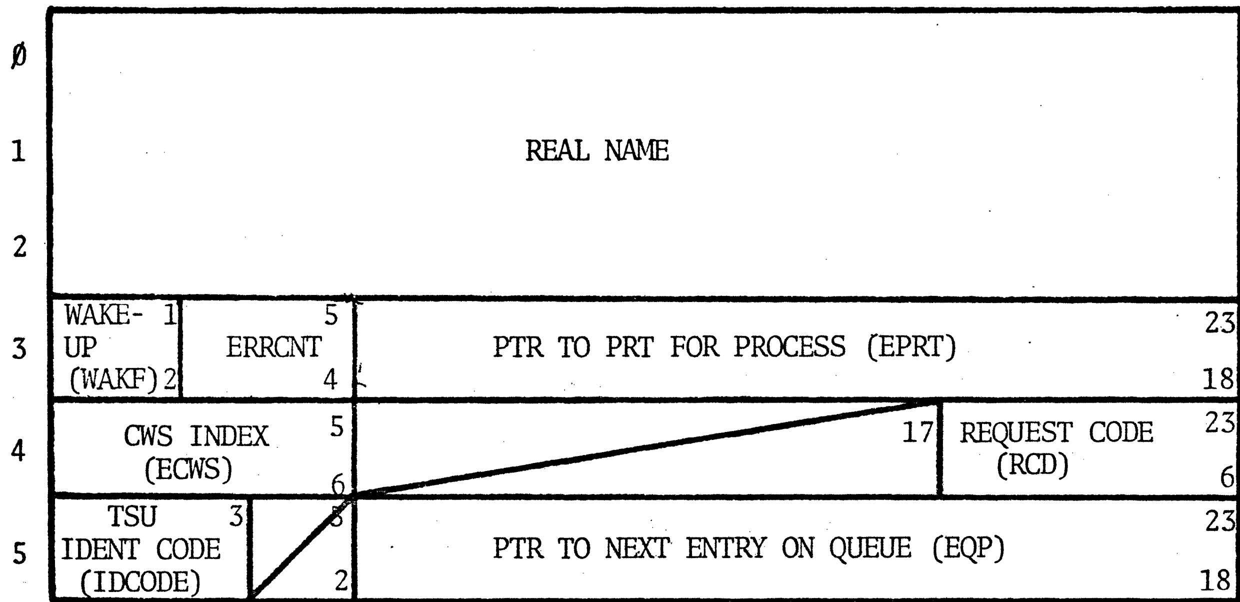


Fig. 4.1 Normal Request Entry

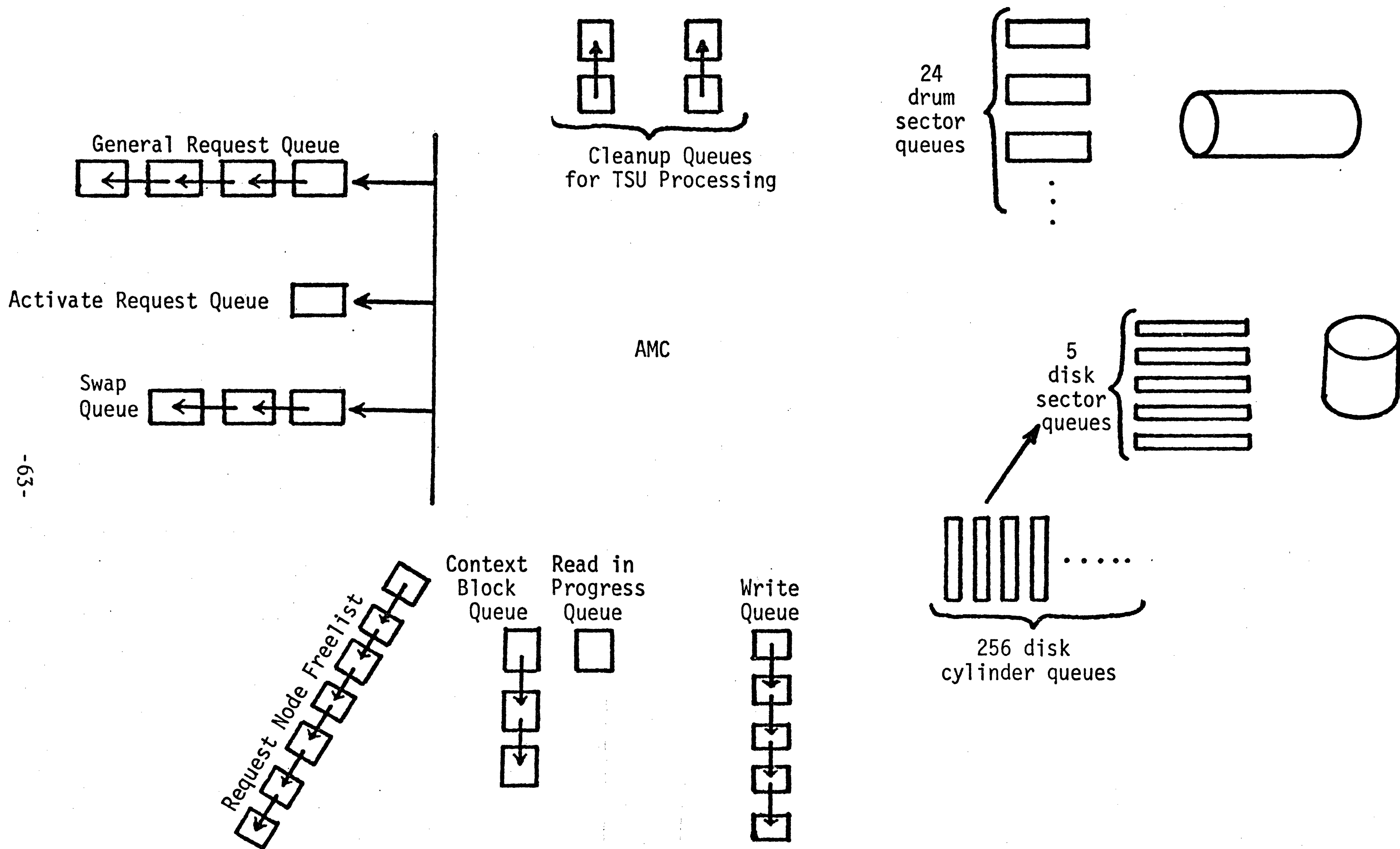


Figure 4.2 Various AMC Queues

There are twenty-four such identically functioned queues, one for each sector on the drum. Depending on which sector of the drum the Context Block is located, (assuming it is on the drum), the request will be placed on the appropriate sector queue. We recall from our previous discussion that the drum signals that it is between sectors (EOS) which allows position monitoring (PC) so that the request can be serviced (on the TSU) when the appropriate sector comes up. As sectors on the drum come up, the corresponding sector queues are searched for requests. Having sent the request (now transformed to a TSU command), the same request is placed on a

CLEANUP QUEUE

There is one such queue for each TSU. After the request has been completed by the TSU at the end of sector time (EOS), the AMC looks at the requests on the cleanup queue. "Cleaning up" includes checking for errors and abnormalities. Assuming the context block was smoothly read in, the cleanup in this case will put the request on a

CONTEXT QUEUE

context block queue. This is a queue of processes whose context blocks only have been read in. In time, the AMC accepts this request. The AMC then looks in the context block for the set of pages that are to be read into central memory for this process (CWS), and a number of requests are generated and placed on the appropriate drum sector queues. Thus if there are requests for two pages on drum sector 2, there will be two requests on the queue for drum sector 2. In this way requests are scattered to the various queues. By doing this, as the drum comes to each succeeding sector, there will be requests queued up for that sector. Thus the drum will be kept busy transferring continuously.

In the same way the context block was read from the drum, as these pages are read, they go onto the cleanup queue, and finally the request node is freed. When the last of the pages is read in, the process is considered to be loaded. The scheduler is then signaled with a wakeup signal that it should now consider the process ready for assignment to a CPU.

In this illustration, one sees the major role played by queues in the implementation.

Figure 4.2 illustrates the various queues that the request node can go onto. Request nodes are removed from the freelist when needed. Requests from the outside world come via the general request queue, activate queue, and swap queue.

When looking at Fig. 4.2, notice that a request node for swapping in moves from the swap queue to the context block queue once its context block has been read in. At this point a whole flock of request nodes are put on the drum sector queues to read in the process' core working set. There is one queue for each sector on the drum, and note, also, one queue for each sector on the disk. The memory manager loads the controller with a command for a sector immediately before the heads pass over it, using the request node from the corresponding sector queue. At this time the node is put on the cleanup queue. Upon successful completion of the transfer, the node is removed from the cleanup queue.

There are 256 disk cylinder queues. The heads on the disk move to each succeeding cylinder in a round-robin fashion. As we move to succeeding cylinders, the entries in the cylinder queues are portioned out to the different sector queues for the disk. Sector queues are reloaded each time the disk moves to a new cylinder.

The write queue holds the requests for those which need to be written onto the drum. In the process of transferring pages, a write is done for a sector if there is no read to be done. When the page is written out, the location on the drum where it had been before is freed. Thus it does not have to return to the same (static) location. If we find that the page needs to be loaded in core, say for another process, then we abort the write and remove the request from the write queue.

This concludes the discussion of the major queues and their uses.

4.1.1 General Request

The strategy for the CPU is to first get a free request node from the free request entry list (FREL) while protected by protect 2.* Then it appends the completed request onto the General Request Queue (the appending operation under protect 2). Finally it sends a request strobe to the AMC. (Note: protects, strobes, etc. are discussed in the section on communications). The general requests are:

Request Code	Function
1	Write process onto drum
4	Direct drum transfer
5	Direct disk transfer
6	Return page to drum

A process may be brought into core by a strategy similar to that of a general request, except that the entry is appended to the Process Input Queue. The request code is 1.

*Protects are described in Section 8.1 on Communications

4.1.2 Activate Request

The activate request combines a data structure with an algorithm to effect a call on the AMC. The basic idea is the CPU will send some data to the AMC and then wait until the AMC replies. The AMC must reply quickly. Furthermore, the AMC may return a word of data which declares what action was taken by the AMC. The CPU then reads the data and then frees the activate port into the AMC. There is only one activate port. It consists of two words for control and information purposes, and two words which are the header of the activate queue.

The CPU strategy is to first lock out all other processors attempting to send activates (by using PROTECT 2). Then check the first Activate Cell (100B absolute). If the cell is non zero, release the protect and try again. Otherwise set the cell to 1 and release the other processors (if desirable). Now put the request on the Activate queue (which should be empty) and send a request strobe to the AMC.

When the AMC has finished, it will have disposed of the entry. It will also succeed or fail return to the CPU. Success is indicated by the number "2" in the first Activate Cell. Failure is indicated by a negative number in the first Activate Cell. In addition, when failure is indicated, the second cell is set to a non-zero value which has the following meaning:

1-TELL CPU TO WAIT, COME BACK LATER

2-UNIQUE NAME ALREADY EXISTS

3-DISK ADDRESS IN USE

4-NOT IN DHT

5-DISK ADDRESS DID NOT COMPARE IN CHT

6-WAIT FOR WAKE-UP

13-ACTIVATE REQUEST OUT OF BOUNDS

For both success and fail returns the CPU will zero the two Activate Cells (100B & 101B).

The activate requests are as follows:

Request Code	Function
11	Transfer page from drum to disk
12	Write Unique Name onto disk
13	Transfer page from disk to drum
14	Make new page
15	Destroy specified page

4.1.3 Request Entry

This is the basic data structure for the AMC queues. It moves about from queue to queue, being modified by the AMC as necessary. A Free List is maintained of all entries not currently being used. The CPU (and others) remove a free node under protect 2. The six words then serve to keep the information as long as the request is in the realm of the AMC. The entry has two forms, one for normal requests and one for Direct I/O requests. Normal requests will contain a subset of the following (see Fig. 4.1):

a. Unique Name

b. Disk address

c. Wakeup condition (Wakeup is sent if field not zero)

d. Error count for re-trying command on device errors.*

e. Pointer to Process Table for process making request.

f. Core Working Set index.*

g. Request code

h. TSU identification code*

i. Pointer to next entry

Direct I/O requires more information to be sent with the original request. It contains the following (see Fig. 4.3):

a. Unique Name

b. Disk or drum address in the compact internal form (which includes the TSU # and Unit #). Described at the end of Section 3.2.1.

c. Wakeup condition.

d. Error count for re-trying command on device errors.*

e. Pointer to the process table for the process making request.

f. Logarithm to the base 2 of the word count. This defines a reasonable set of word counts: Ø, 1, 2, 4, ..., 2^{23}.

g. Core page number in absolute core.

h. Timing (i.e., when to send request: Correct, ahead by one sector, behind one sector, or half drum revolution away from correct time).

i. Dump the TSU state at the end of the instruction.

j. Recover if an error was made during the transfer by doing the transfer again.

*These fields are not set by CPU.

k. Request code.

l. TSU instruction exactly as sent to the TSU.

m. Pointer to next entry.

4.2 The Core Hash Table

4.2.1 CHT Layout

Information about the current contents of core memory is maintained in the core hash table, which is a chained hash table using the unique name as key.

<table>
<tr><td>Ø</td><td colspan="7">0 — UNIQUE NAME — 23</td></tr>
<tr><td>1</td><td colspan="7">0 — UNIQUE NAME (cont.) — 23</td></tr>
<tr><td>2</td><td colspan="7">Drum or Disk Address (Internal Form)</td></tr>
<tr><td>3</td><td>WAKE-UP (WAKF) 1 2</td><td>ERRCNT 5 4</td><td colspan="5">PTR TO PRT FOR PROCESS (EPRT) 23 18</td></tr>
<tr><td>4</td><td>LOG_2 WORD COUNT</td><td>CORE PAGE (PGAD) 12 7</td><td>/</td><td>TIMING 15 2</td><td>DUMP</td><td>RCVR</td><td>Request Code (RCD) 23 6</td></tr>
<tr><td>5</td><td>TSU Instruction</td><td colspan="6">PTR TO NEXT ENTRY ON QUEUE (EQP) 23 18</td></tr>
</table>

Fig. 4.3 Direct I/O Request Entry

The table comes in two parts:

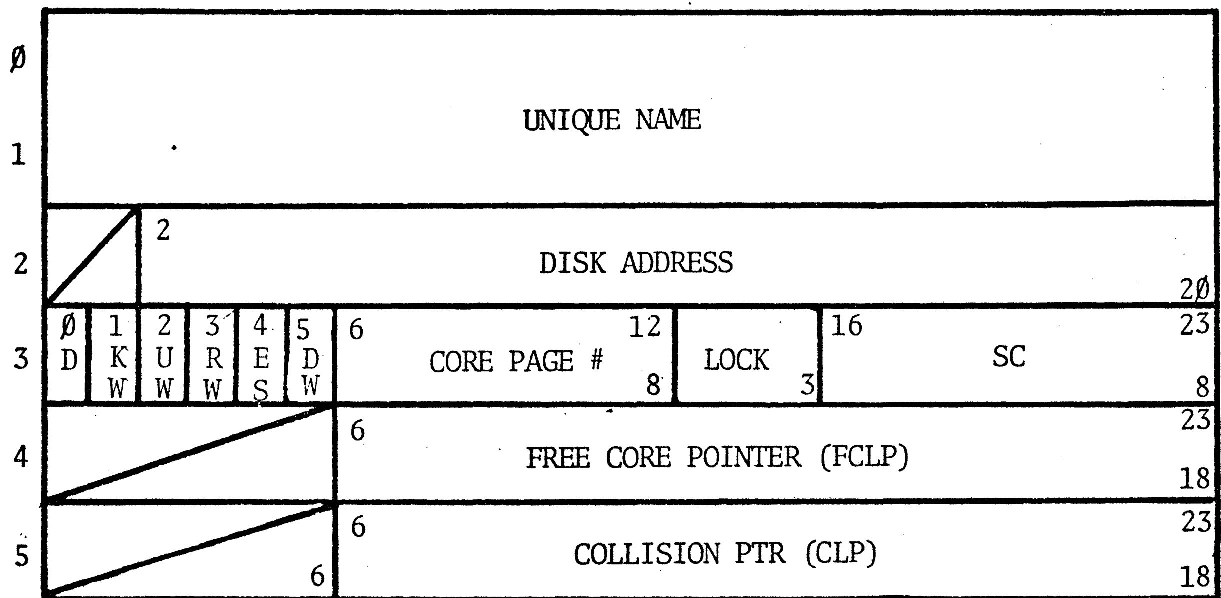
1) The index, called CHT1, which is an array of 256 pointers to lists of CHT entries. Each word of CHT1 is either END or the address of a CHT2 entry with the property that HASH (UN(e)) is the address of the CHT1 word. If there are several pages in CHT with the same value of HASH (UN), the CHT1 word points to one of them, which points to the next using the collision pointer field, and so on until all are chained onto the list. The last one has END in its collision pointer. The ordering of pages on this chain is not significant. Since there are only 64 core pages, chains of length >1 should be infrequent. The hashing function HASH is to take the exclusive OR of the 6 8-bit bytes of a UN and then the exclusive OR of this result with 364B. END is the standard end-marker for chains in the MMS: it is 777777B in the last 18 bits of a word; the first 6 bits are ignored.

2) The body, called CHT2, which is an array of NCORE entries of 6 words each. The index of an entry in CHT2 gives the physical page it describes. The format of an entry is given in Fig. 4.4.

4.2.2 When a Page is 'in CHT'

A page is said to be in CHT if it can be found by the hash table search. If a page is in CHT, it is guaranteed to be the page specified by the UN in CHT and if it got there by a drum or disk read, there was no detected error in the transfer. To put this more precisely, there are three ways for a page to get into CHT:

1) by a drum read in which the UN in the PMT entry which gave rise to the read agrees with the UN read from the drum and no error was detected during the transfer.



D = DIRTY

U = UNAVL, unavailable to CPU

DW = DWIP, drum write in progress

KW = disk write

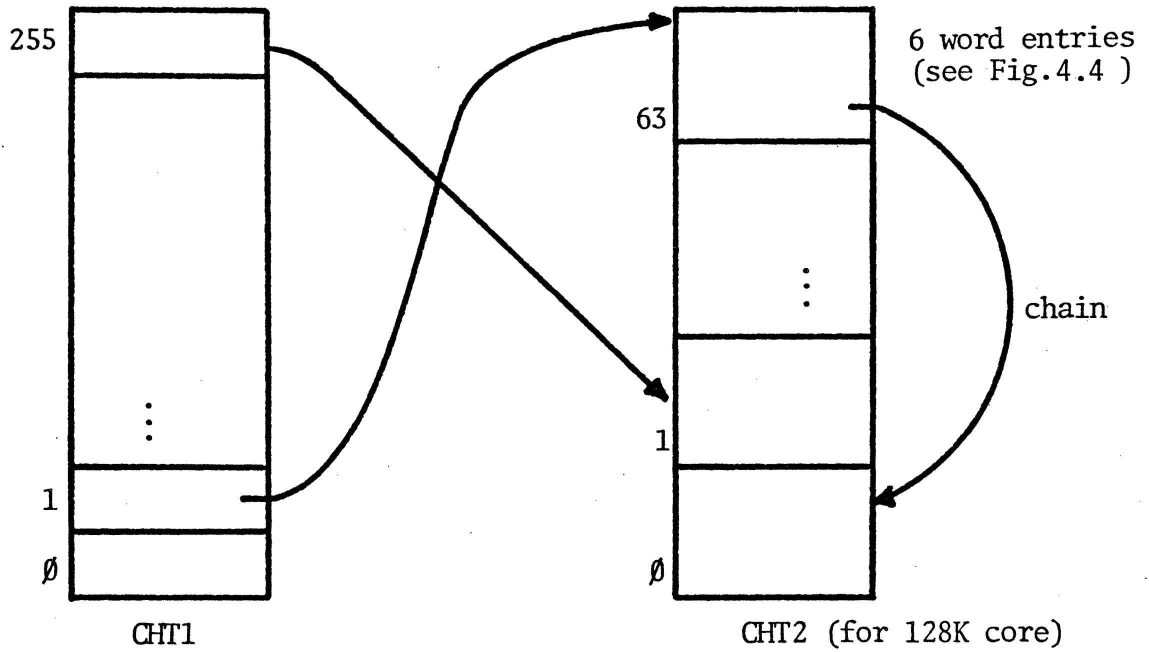
E = read error

SC = scheduled count

R = RIP, read in progress

The free core pointer holds the page on the free core list. The collision pointer holds the page on the CHT hashing structure.

Figure 4.4 Core Hash Table (CHT2) Entry



The figure shows physical pages ∅ and 63 with hash code 1, physical page 1 with hash code 255

Fig. 4.5 CHT Hashing Structure

2) by a disk read in which the UN in the read request agrees with the UN read from the disk and no error was detected during the transfer.

3) by creation of a new page. In this case the UN is the one given in the create request.

In cases (1) and (2) the page goes into CHT when the read has been successfully completed, in case (3) when the request is satisfied.

Recall that every physical page has an entry in CHT2. This entry describes the status of the page whether or not the page is in CHT. Pages not in CHT are of the following types:

1) empty pages. These come into existence only at initialization, when a page is destroyed, or when a read fails.

2) pages involved in a read actually <u>in progress</u> from drum or disk. A page is allocated for this purpose when the read is given to the TSU. At the completion of the transfer it is either put in CHT (if the read was successful according to (1) or (2) above) or becomes empty. A page cannot remain in this state for more than 2 disk sector times. A page has the RIP bit set if and only if it is in this state.

4.2.3 <u>CHT Page Status Information</u>

In addition to the RIP bit mentioned above, several other bits record the status of a CHT page. They are all quite independent, except as stated.

1) DWIP is set when a drum write is in progress. It is exactly analogous to RIP. When DWIP is set the page becomes clean. When DWIP is cleared the page goes onto the free core list, unless the write fails.

2) KW is set when the page is brought into core to be written on the disk. It remains set until the disk write is successfully completed or abandoned. Note

that it is not analogous to DWIP.

3) DIRTY is on when the page in core is potentially different from the copy on drum. DIRTY is set by the CPU whenever it does a store. DIRTY is cleared when a drum write is started, at the time DWIP is set. If DIRTY is off, the page is said to be clean; a clean page with Ø scheduled count may be removed from CHT and allocated to some other purpose.

4) UNAVL prevents CPU access to the page. It is never changed or referenced by the MMS.

5) LOCK prevents the page from being removed from CHT if it is ≠ Ø. It is not set or cleared by the MMS. It must not be cleared when SC=Ø, or the page removal algorithm will not work.

4.2.4 Scheduled Count (SC) and Accessibility of Pages

The SC field of CHT counts the number of loaded context blocks in which the page appears in the core working set and has SF on, plus 1 if KW=1. It serves two purposes:

1) A page should be removed from core when there are no loaded working sets which refer to it. Since the SC field counts the number of loaded working sets referring to the page, the right time to remove the page is when SC=Ø.

2) When a page is read from drum or disk, a comparison of the recorded UN with the requested UN is always made. When a page is in core and being referenced by a CPU, on the other hand, its physical core address sits in the physical map (PM)* and is used directly; no check for the proper UN is made when

*The physical map is a map between physical and virtual pages kept for each CPU.

the page is referenced. It is therefore important to be able to control CPU access to a page and to know whether a CPU could have the page in its PM or not.

We will say that a CPU <u>can access a page</u> if the page (specified by its UN) is in CHT, UNAVL=Ø, and the SF bit in the CPU's PMT entry =1. The CPU will load the physical address of a page into its PM only if it can access the page. Once the page is in the PM, of course, the CPU can reference the page regardless of the state of CHT. The map-clearing machinery described below permits control to be exercised over a page which is in a PM.

We now observe that if SC is correct there cannot be a CPU which can access a page with SC=Ø, since for such a page there is no PMT byte with SF=1. This means that if SC=Ø a page can be removed from CHT (or its DIRTY bit can be cleared) with the assurance that any subsequent attempt to reference the page from a CPU will not find the page (or will discover that the DIRTY bit is Ø), since any CPU reference to a page not in the PM must go through CHT.

The SC is maintained in the following way:

1) When a context block has been read, the working set is scanned. Each page is looked up to see if it is in CHT. If it is, SC for the page is incremented and the SF bit is set. Otherwise a read is queued for the page. No account is taken of other reads which may be queued for the same page (at least not for the purposes of this discussion).

2) When the read is considered (for details on when this happens see the discussion of swapping below) a second attempt is made to find the page in CHT. If it is found, again SC is incremented and the SF bit set, and the read is abandoned. Otherwise the read is performed. If it is successful the page is put into CHT, its SC is set to 1 and the SF bit set.

3) When a process is written out, the working set is scanned again. For each page with SF=1, SC is decremented by 1 and SF set to Ø.

4) When a page is released to the drum, its SF bit is cleared (the request is an error and is aborted if SF=Ø) and SC decremented.

4.2.5 Removing a Page from CHT, or Clearing DIRTY

These operations can be performed only when SC=Ø, hence only when no CPU can access the page. Since there is no entry in any PM for this page, all that is required is to remove it from the hashing list structure, or to reset DIRTY.

Since the consequences of a mistake at this point are extremely serious, however, a very powerful (and expensive) cross-check is provided to ensure that a page being emptied or cleared is in fact not in any PM. The MMS has an operation called selective map clearing which puts both physical maps in a state which will cause the CPU to perform a local selective map clearing the next time the map is accessed. The local clearing proceeds as follows:

1) The CPU picks up from a fixed core location peculiar to it a physical page number. This number is set up by the MMS to the number of the page being emptied or cleared before the MMS does the selective map clearing. When setting up the location, the MMS waits for it to become Ø first.

2) The CPU then tells the PM to scan for a non-empty entry containing the specified physical page number.

3) If no such entry is found, the CPU zeros the core location and proceeds on its way.

4) If the entry is found, the page being emptied or cleared was in the PM. This indicates a failure of the MMS. The CPU empties the PM entry and proceeds with the local clearing. When it is finished, the CPU generates a fixed trap to the monitor, which can cope with the situation as it sees fit. In this case the core location is left untouched so the monitor can find out what happened.

The procedure just described ensures that the PMs cannot get out of touch with the CHT. It is carried out immediately after a page is removed from CHT or the DIRTY bit reset, and before any action is taken which depends on the fact that the page cannot be referenced. Once the selective map clear is done, the amount of time which must elapse before the page is completely safe from CPU references is the largest time which can elapse in the CPU between a reference to the map and the use of a mapped address (not necessarily the same one).

The cost of a selective map clear is about 10μs of CPU time. It is necessary to do one for each page transferred to or from the drum: in the case of a read, because the transfer requires freeing a physical page, and in the case of a write because DIRTY must be cleared.

The contents of this section so far imply that sneaky writes (writes of dirty pages which are still in use) will never be done, and this is indeed the intention. It is, however, possible to handle sneaky writes at the expense of complicating the trap routine in the monitor which receives the trap caused by doing a selective map clear while the page is actually in a PM.

The MMS keeps a <u>free core list</u> (FCL) of pages which are candidates for reallocation. A page is put on this list, provided it is not there already:

1) When its SC is decremented to Ø, LOCK=Ø, its drum address in DHT is not Ø and its DIRTY bit is Ø.

2) When a write for the page is successfully completed.

When a core page is needed (for a drum or disk read, or to satisfy a create new page request) one is taken from the FCL and treated as follows:

1) If SC≠∅, it is ignored. This means that it has shown up in a core working set being loaded since it was put on the FCL, which is quite possible. If LOCK≠∅, it is also ignored.
2) If DIRTY≠∅, it is ignored. Cf (1).
3) If LOCK≠∅ or RIP≠∅ or KW≠∅, punt. This is a MMS error, since SC should never be ∅ when the page is locked or being written on the disk, and a page being read into should not be in CHT at all.
4) Otherwise it is removed from CHT and a selective map clear is done for it. The physical page may then be put to its new use.

This is the only way in which a page may be removed from CHT.

4.3 Drum Hash Table

4.3.1 General

Information about the current contents of the drum is maintained in the drum hash table, which is a linear hash table using the disk address as key.

The basic information in a DHT entry d is

1) a disk address K(d), used to search the table
2) a drum address D(d) which is the current drum location of the page at the given disk address
3) a use count UC(d) which more or less counts the number of processes in which the page appears in the DWS.

4.3.2 DHT Page Status Information

In addition to K, D and UC, a DHT entry contains some additional information which keeps track of the status of the page.

1) ONDK (On Disk), set if the disk copy of the page is valid. This bit is set after a successful disk read when a DHT entry is made, and just before a write. It is cleared when a page is created, after an unsuccessful write and after a drum write if DIRTY is set. Note that the first drum write for a page newly read from disk normally is done because ONDR (see below) is ∅ and not because DIRTY=1; this write does not clear ONDK, which is what we want.

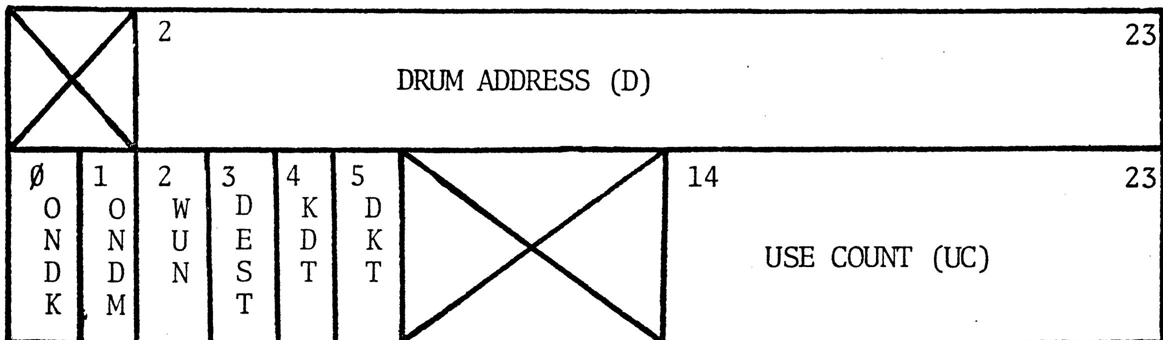
2) ONDM (On Drum), set if a copy of the page exists on the drum. This bit is set when the page is written onto the drum. It is reset only when a DHT entry is newly created for a page which is being read from disk to drum or for a page which has just been created in core. ONDM=∅ causes the page to be written out of core onto the drum even though DIRTY=∅ in the CHT entry for the page.

3) WUN, set if the drum unique name should be written to the disk. This bit is set only by creation of a new page, and is cleared when the write is successfully completed.

4) DEST, set if a destroy request has been made for the page. This bit can only be cleared by deletion of the DHT entry.

5) DKT, set if a disk write is going on. This bit is set when the decision is made to write, i.e., when the write request is put on the disk queues. It is cleared when the write is completed successfully.

DHT is organized as two parallel tables called DHT1 and DHT2. DHT1 is the linear hash table and has 1 word per entry, containing the disk addresses. DHT2 is maintained parallel to DHT1 and has 2 words per entry, containing all the other information about the page in the following format.



ONDK - On Disk

ONDM - On Drum

WUN - Write Unique Name

DEST - Page is being destroyed

KDT - Disk to Drum Transfer in progress

DKT - Drum to Disk Transfer in progress

Figure 4.6: Drum Hash Table Entry

4.4 Standard Circular List Structure

The AMC must keep a lot of data in queues and stacks and lists. It was considered best to have one mechanism for manipulating all of these structures. Therefore a circular list structure was developed. It has the following properties:

a. A node may be attached to the front of the list preceding all other nodes. This operation is known as stacking the node.

b. A node may be attached to the end of the list behind all the other nodes. This operation is known as appending the node to the list.

c. A node may be removed from the front of the list. This is equivalent to unstacking the node if the entries are put on the front exclusively. It is equivalent to removing a node from a queue if the nodes are put on the end exclusively.

d. Only 18 bits of the node are used for the pointer, and the pointer may be in any one of the words in the node.

e. A special pointer (777777B) marks the end of the list.

Each list must have a fixed starting point. This point is known as a "header". In this design the header is composed of two words. The first word points to the front of the list. The second header word points to the last node on the list. Each node contains one pointer for the list which points to the first word of the next node (see Fig. 4.7a). The pointer may be offset from the beginning of the node by any amount. As a natural consequence of the data structure, the empty list has a header with the first word containing 777777B. The second word of the header points to a pseudo-node in which the first word of the header is in the same position as the pointer for that list (see Fig. 4.7b).

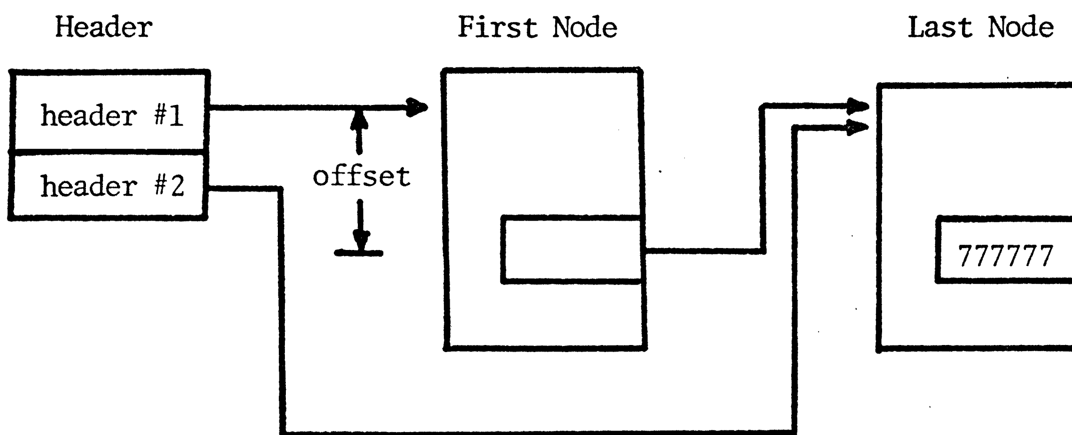


Fig. 4.7a Circular List Structure

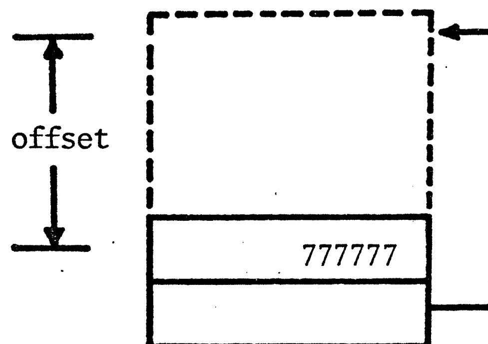


Fig. 4.7b Empty List

4.5 State of AMC Microprocessor

The state of the microprocessor is loaded and stored when crashes and breakpoints occur. Several memory locations participate in this operation. The state is stored into cells 2500B to 2611B in the following format:

2500	SKØ (M)
2501	SK1
⋮	
2577	SK63
2600	RØ
⋮	
2606	R6
2607	OS
2610	Q
2611	Z

Of course SKØ is the contents of M since this scratchpad register is reserved for saving M. When the state has been saved, the cell Breakwait is tested. For the AMC this is cell 25B. The address of the instruction to execute after the state has been restored is kept in Break (20B).

4.6 Important Central Memory Locations

Since the system has many processing units, each needing some information which is a subset of all the information needed to operate the system, all the system tables are in main memory. The AMC also requires information which is

considered private to itself (although certainly diagnostic or recovery routines may wish to look at it). In phase 1 the private tables are also kept in main memory.

The following is a map of the interesting memory:

5	SYSTEM RESTART REGISTER
20	BREAK ADDRESS
25	BREAKWAIT CELL
100	ACTIVATE CELL
101	ACTIVATE CELL
102	ACTIVATE QUEUE HEADER START
103	ACTIVATE QUEUE HEADER END
104	GENERAL REQUEST QUEUE HEADER START
105	GENERAL REQUEST QUEUE HEADER END
106	PROCESS INPUT QUEUE HEADER START
107	PROCESS INPUT QUEUE HEADER START
110	FREE REQUEST ENTRY LIST HEADER START
111	FREE REQUEST ENTRY LIST HEADER END
112	FREE CORE LIST HEADER START
113	FREE CORE LIST HEADER END
114	PROCESS READS IN PROGRESS HEADER START
115	PROCESS READS IN PROGRESS HEADER END
116	CONTEXT BLOCK QUEUE HEADER START
117	CONTEXT BLOCK QUEUE HEADER END
120	WRITE QUEUE HEADER START
121	WRITE QUEUE HEADER END

122 NUMBER ON FREE CORE LIST

123 NUMBER ON FREE REQUEST ENTRY LIST

124 ERROR PROCESS FOR SWAPPER

125 COUNT OF TIMES COULD NOT REMOVE NODE FROM FREL

126 CLEANUP BUFFER HEADERS

135 CLEANUP BUFFER HEADERS

136 DISK SECTOR QUEUE HEADERS

⋮

151 DISK SECTOR QUEUE HEADERS

152 NICT ERROR COUNT

153 NUMBER OF CONTEXT BLOCKS LOADED

154 NUMBER OF PROCESSES BEING LOADED

155 BASE ADDRESS OF DRUM SECTOR READ LIST (DSRL)

156 BASE ADDRESS OF DRUM FREE PAGE TABLE (DFPT)

157 BASE ADDRESS OF DISK CYLINDER QUEUES (KCQ)

400 - 777 CHT1

1000 - 2377 CHT2

PART OF STATE OF MICROPROCESSOR

2517 BASE ADDRESS OF DHT1

2520 BASE ADDRESS OF DHT2

2521 SIZE OF DHT

4.7 Statistics (Counters)

COUNTERS

THE FIRST BLOCK IS COUNTS FOR VARIOUS TYPES OF I/O TRANSFERS INITIATED (STARTUPS)

200B CONTEXT BLOCK

201B PAGE READ (BOTH CONTEXT AND PROCESS PAGE)

202B DRUM READ FOR DRUM TO DISK TRANSFER

203B DIRECT DRUM TRANSFER

204B DIRECT DISK TRANSFER

205B WRITE FOR PAGE NOT ON DRUM

206B WRITE FOR DIRTY PAGE

207B CHECK UN ON DRUM FOR DESTROY PAGE

210B NOTHING

211B WRITE ON DISK FOR DRUM TO DISK TRANSFER

212B CHECK UN ON DISK FOR DRUM TO DISK TRANSFER

213B CHECK UN FOR WRITE UNIQUE NAME ON DISK

214B READ DISK FOR DISK TO DRUM TRANSFER

215B WRITE FOR WRITE UNIQUE NAME

216B CHECK UN ON DISK FOR DESTROY PAGE

217B DESTROY ON DISK

220B READ DRUM FOR DISK TO DRUM TRANSFER

221B READ PAGE KLUDGE

222B PAGE FAULT READ

223B PAGE FAULT WRITE

224B PAGE FAULT WAKEUP

THE NEXT BLOCK OF COUNTERS ARE FOR COUNTING ACTIVATE ERROR RETURNS

IT BEGINS AT 225B

225B WAIT CPU

226B UNIQUE NAME ALREADY EXISTS

227B DISK ADDRESS IN USE

23ØB NOT IN DHT

231B DISK ADDRESS DID NOT COMPARE IN CHT

232B WAIT FOR WAKE-UP

233B DISK ADDRESS REASSIGNED

237B ACTIVATE REQUEST OUT OF BOUNDS

24ØB NO REQUEST NODE

241B NO CHT ENTRY

242B SCHEDULED COUNT OVERFLOW

243B PROCESS IN TROUBLE

THE NEXT BLOCK OF COUNTERS ARE FOR COUNTING 'AMC' ERRORS

IT BEGINS AT 246B

246B REDO WRITE FOR ALL REASONS

247B TAG FIELD ERROR

25ØB REGISTER LOADING VIOLATION

251B TOTAL DEVICE ERRORS

252B RETRY COMMAND

THE NEXT BLOCK OF COUNTERS ARE FOR COUNTING THE NUMBER OF ERRORS SENT TO A PROCESS. IT BEGINS AT 251B.

AMC FAIL RETURNS TO CPU

251	CPWAIT	TELL CPU TO WAIT, COME BACK LATER
252	$UNER	UNIQUE NAME ALREADY EXISTS
253	DKINU	DISK ADDRESS IN USE
254	NINDHT	NOT IN DHT
255	DNCHT	DISK ADDRESS DID NOT COMPARE IN CHT
256	WAITWK	WAIT FOR WAKE-UP
257	DKRAS	DISK ADDRESS REASSIGNED
26Ø	HERDR	HARD ERROR ON DRUM READ
261	SERDR	SOFT ERROR FROM DRUM
262	$HERDK	HARD ERROR ON DISK
263	$UNERK	UNIQUE NAME ERROR ON DISK AFTER DRUM COMPARED
264	$SERDK	SOFT ERROR ON DISK READ
265	$ARQOB	ACTIVATE REQUEST OUT OF BOUNDS
266	NOREQ	NO REQUEST NODE
267	NCHT	NO CHT ENTRY
27Ø	SCHOVF	SCHEDULED COUNT OVERFLOW
271	PRTRBL	PROCESS IN TROUBLE

4.8 Table Manipulations

Time is not measured in milliseconds in the swapper but rather by relation to the events taking place. The following times are the important times in referencing events:

1) The time when a request is considered. Let me call this Request Time (RQT).
2) The beginning of a transfer, i.e., a node is removed from a queue and commands sent to the TSU. Let me call this Start-up Time (STPT).
3) The end of a transfer when the node is again perused to determine what to do next. Let me call this the End of Transfer Time (EOTT).
4) The time that a process' context block is scanned for the reads or writes. This occurs as frequently as required by the swapper to maintain its flow. Let me call this Process Scan Time (PSNT).

Core Hash Table

Dirty bit:

Set - by someone when page modified.

Reset - Start-up of write to drum.

Disk Write bit:

Set - (STPT) immediately before disk write command sent to disk TSU.

Reset - EOTT of disk write.

Tested - When trying to determine if page is "in core".

Page Status (4 bit field):

Set - Start-up of all transfers.

Set - EOTT and error in transfer.

Clear - At end of all transfers if transfer is successful.

Scheduled Count:

Inc. - Context block read - at RQT if context block in core, otherwise at EOTT.

Inc. - Page of a process - at time context block is scanned if page in core, otherwise at EOTT.

Dec - Context block read - after context block is scanned and it is put on the Read in Progress Queue since it is in the Core Working Set.

Dec - Page of a process - RQT, when context block scanned for pages for a write process request.

Dec - Page of a process - RQT, for return page to drum command.

Free Core Pointer:

Set - When it is determined that a page must go onto the free core list.

Reset - When it is determined that a page may be removed from the free core list.

Page Lock:

Set - ? Not by Swapper

Reset - ? Not by Swapper

<u>Drum Hash Table</u>

Disk address (field):

Set - when DHT entry made

Cleared - when DHT entry deleted

Location (two bit field):

Set - (EOTT) end of write onto drum

Set - (EOTT) end of write onto disk

Clear - (RQT) request new page

Write Unique Name:

Set - (RQT) request new page

Reset - (EOTT) transfer from drum to disk

Reset - (EOTT) write Unique Name onto disk

Destroyed:

Set - (RQT) delete page on disk

Disk-to-Drum Transfer in Progress:

Set - (RQT) disk-to-drum transfer

Set - (RQT) write Unique Name onto Disk

Reset - (EOTT) either transfer completed

Drum-to-Disk Transfer in Progress:

Set - (RQT) drum-to-disk transfer

Reset - (EOTT) transfer completed

Unique Name Check:

Set - (RQT) destroy page

Hard Drum Error on Read for Destroy Page:

Set - (EOTT) destroy page

Use Count (field):

Inc - (RQT) transfer page from disk to drum

Set - (RQT) request new page

Dec - (RQT) transfer page from drum to disk.

Request Node

Error Count:

Inc - (EOTT) when an error is detected in a transfer.

Reset - (EOTT) when transfer is successful.

CWS Index:

Set - (PSNT) when context block is scanned to queue reads.

Identification Code:

Set - (STPT) when TSU given instruction to aid in identifying the entry in the buffer.

Request Code:

Set - (EOTT) when necessary to requeue request as another request in sequence.

Process Memory Table

Class Code Error:

Set - (EOTT) if class codes do not match on read for a process.

Scheduled Count Flag (SF):

Set - (PSNT) when context block is scanned to queue reads.

Reset - (PSNT) when context block is scanned to queue writes.

Reset - when there are hard errors.

Process Table

AMC Interrupt Bit:

Set - (EOTT) whenever the wakeup is requested in the original request.

Swapper Error Word:

Set - (EOTT) a hard error occurs.

Drum Transfer Count:

Inc - when the context block is scanned and a read is queued.

Dec - (EOTT) a process read is successful.

Processed Queued:

Set - when process is put on Read In Progress Queue

Reset - when process loaded

Swapper Queue:

Set - SWQ is set by μ-Scheduler or CPU when they send a SWAPIN request for the process to the AMC.

Reset - (RQT) when taken off queue for context block read.

Loaded:

Set - when process in core and wakeup generated.

Reset - LDD is reset by μ-Scheduler when it sends a SWAPOUT request for the process to the AMC.

5. INTRODUCTION -- SOFTWARE AND FIRMWARE

Having looked at the hardware aspects of the system, and quite a bit of data structures, we come now to the actual code in the system. As has been alluded to earlier, we have both microcode and an implementation language (assemblyish code) called APU code. The implementation language instruction set is actually implemented in microcode. This overall organization is shown in Fig. 5.1.

The obvious advantage of microcode is its high computation rate. The microprocessor cycle time is 100 ns. Because we are using ROM, the disadvantage is difficulty in modifying the code. The APU code has an inverse characteristic, easily modifiable, but considerably slower than microcode.

What has been coded in microcode are those functions that are well-defined and called frequently. Most primitive functions are microcoded. Coded in APU code are the more policy dependent, less often called functions. Because this was a first attempt at putting memory management into a separate processor, it was not clear as to what should go into microcode and APU code. A proposed later version of the memory manager, in fact, puts into microcode many APU functions of the present version.

The subroutine mechanism of the memory manager is a powerful and flexible one. As usual the APU code has a subroutine call mechanism to other APU routines. But it also has a subroutine call mechanism to call microcode subroutines. Likewise microcode routines can call other microcode routines. The mechanism is not fully symmetric in that microcode cannot call APU routines (though it can branch to them). Using this structure, the memory manager code is a powerful mix between microcode and APU code. APU routines make use of calls to very fast microcoded routines to perform many tasks.

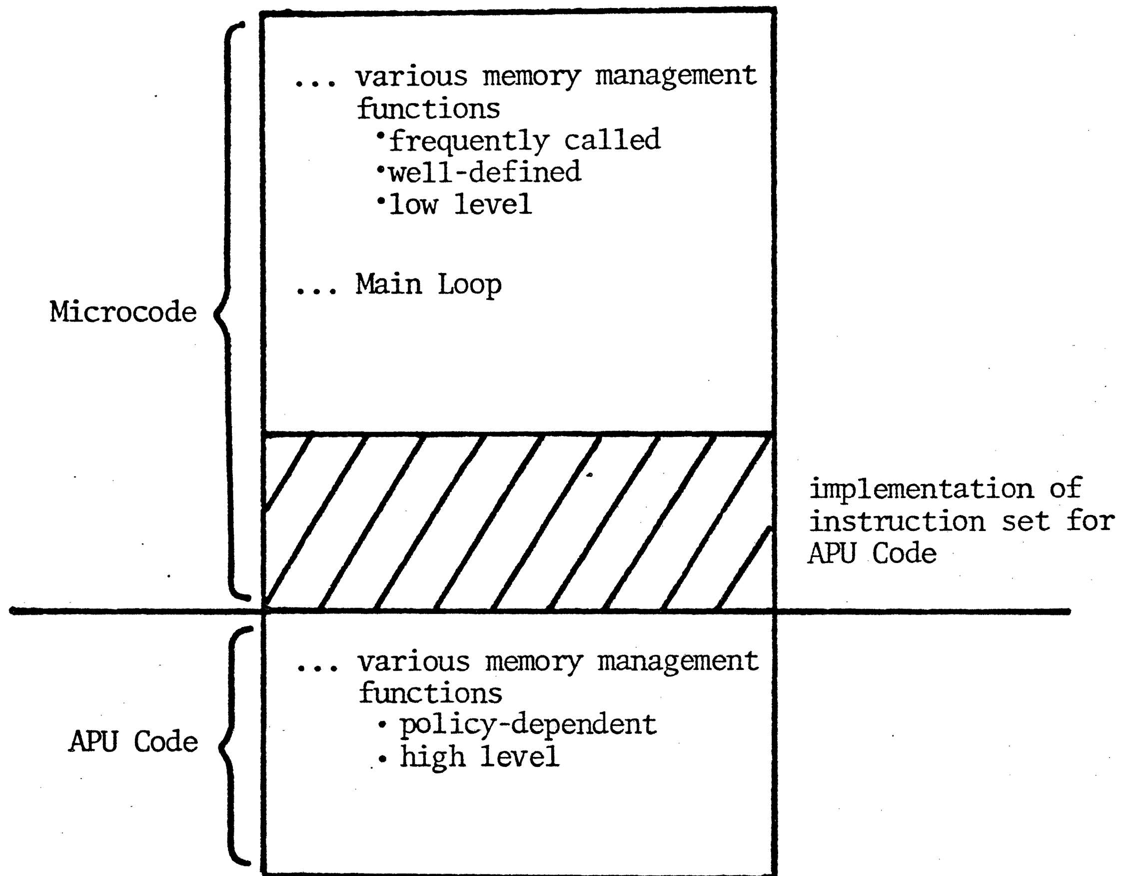


Fig. 5.1 AMC Code Organization

It should be pointed out that in order for the memory manager to perform effectively, it must do its work subject to various time constraints. For example, between the time of one end of sector (EOS) when the registers swap (HR and FR, see Fig. 3.7) and next EOS, the memory manager must analyze the results of the last swap, check for and handle errors, and also set up the transfer for the next sector to come. This it must do for each TSU. On the side, it must find time to read context blocks and queue up page reads, move the heads on the disk, etc. It is clear why CPUs are not assigned this type of memory management function. They would never have time to run user processes! The use of microcode is very important to the effectiveness of the system. Poor performance is marked by less than continuous swapping of pages (that is, if there is enough work) mainly attributable to the memory manager not being able to handle its workload, i.e. being too slow.

The first of the two following sections describes the microcode. We turn first to the main loop, for this is where the memory manager starts. From here it dispatches to various microcode and APU routines.

6. MEMORY MANAGER FIRMWARE

This section describes the firmware portion of the memory manager.

6.1 The Memory Manager's Main Loop

Figure 6.1 shows the main loop, the center of activity for the memory manager. One of the basic activities of the main loop is checking the attention flag. TSUs set this flag when they come to their EOS's. Notice in the main loop the various places that ATTENTIONS are checked for.

Receiving an ATTENTION we look for CLEANUPs that need to be done. As discussed in the data structures (Section 4.1), there is a cleanup queue for each TSU. The memory manager checks to make sure that all went well in the last transfer, taking appropriate action for error cases. Depending on what the request code was (Fig. 4.1 shows request code as part of request entry), we dispatch to one of several APU cleanup routines. For example, the cleanup for a context block read involves setting various flags that a cleanup for a regular page read doesn't.

Having done all possible cleanups, we then do STARTUPs for the various TSUs. This involves packing the various TSUs with the correct transfer commands for the forthcoming sector. Microcode functions search the appropriate drum/disk sector queue for the node and then dispatch to one of several APU startup routines.

The different cleanups and startups will be discussed in more detail in the section on APU code.

If we have time, we try to queue up page reads of the CWS for a context block that has been read in, and then try to read in a new context block for another process.

Much of the memory manager's attention is devoted to simply managing the just mentioned activities.

Other than ATTENTIONS the memory manager also handles activate and general requests. The request flag is set by the monitor or another processor to signify a request for the memory manager. We dispatch to an APU routine via a general request transfer vector or activate request transfer vector.

If at this point there is no ATTENTION, we move the disk to the next cylinder (round-robin fashion) so that transfers for that cylinder can commence.

We continue to service general and activate requests until an ATTENTION comes.

Note that flag-setting rather than interrupt is used to communicate to the memory manager. The assumption is that the speed at which the memory manager does its assignments permits attentions and requests to set flags which will be serviced in an appropriately short time.

Note also that the queue structure allows the memory manager to perform its work in small packets. There are many places in which the memory manager can be diverted to other tasks because of a signal coming in (e.g. ATTENTION). This can be done because the state is maintained in the request nodes and passed from queue to queue.

Thus we have the organization of the main loop. It is mainly a dispatcher to various discrete routines that will finish their tasks quickly and return to the main loop.

6.2 Microcode Descriptions

Following is a list of microcode routines and structures given in chronological order as they occur in the microcode.

The presence of patches makes certain parts a little messy. However, a

general perusal of the microcode gives one some insight into the microcode part of the memory manager.

More detailed descriptions of routines marked with an asterisk are given in the sections indicated.

Loc	Routine or Structure	Name
0	When the Zap signal is sent, the APU is breakpointed with state saved if desired, or else initialized	SWAPR
2	Part of APU main loop	
4	Field logic for APU Code	R2PNT
45	Punt (fail) return setup for AMC	
46	Save State of AMC *(6.3.19)	SAVST
64	Load State of AMC	LOADST
102	Exchange scratchpads with memory	XSKPD
107	Stack Link Subroutine; saving return points, state, for subroutine calls *(6.3.1)	STKLK
121	Subroutine Call, Return Mechanisms	
141	PUNT(fail) logic	PNT
143	CHT Search *(6.3.6)	CHTSCH
156	Enter CHT Entry *(6.3.7)	ECHT
171	Put Page on Free Core List *(6.3.11)	PPFCL
200	CHT Hashing *(6.3.5)	CHTHSH

Loc	Routine or Structure	Name
213	Delete CHT Entry *(6.3.8)	DCHT
222	Clear CHT Entry *(6.3.9)	CCHTE
235	Get Free Core *(6.3.10)	GFC
270	DHT Search *(6.3.13)	DHTSCH
311	Make DHT Entry *(6.3.14)	EDHT
321	Delete DHT Entry *(6.3.15)	DDHT
340	Append Entry to List *(6.3.16)	AEL
347	Stack Entry on List *(6.3.17)	SEL
357	Remove Entry from List (1) *(6.3.18)	RTEQP
362	Remove Entry from List (2)	REL
400	APU Instructions Table	
640	Fail Return Table for Calls	FTABLE
665	APU Instruction Fetch Main Loop	ILØ
725	Area for APU instructions that take more than the table to implement	
752	Call Micro-code subroutines (from APU)	CALL
763	Return from Routine Fail Return from Routine	CALL3
771	Patch Space	
1000	MAIN LOOP *(6.3.2)	MAIN

Loc	Routine or Structure	Name
1023	Copies from cylinder to sector queues for disk	COPYK
1027	Process Attentions *(6.3.3)	PATTN
1033	Call APU Cleanup Routine-part of attention processing	PATN2
1045	Search for TSU needing cleanup	PATN1
1057	Search for TSU needing startup	PATN3
1076	Start APU routine *(6.3.4)	SAPU
1103	Dump TSU state *(6.3.20)	DPTSU
1125	Generate Wakeup	WAKUØ
1141	Send TSU Instruction *(6.3.22)	STSUI
1160	Various test routines	
1175	Stack Entry on Free List *(6.3.23)	SETFL
1202	Remove Entry from Free List *(6.3.24)	REFFL
1216	12-Bit Multiplication Routine	MVLP
1220	Initialization sequence *(6.3.25)	GO
1277	Get Selected Position *(6.3.27)	FSTR
1333	Get Position *(6.3.28)	GETPOS
1346	Continuing Startup, decide to go to STDRUM (1460) or STDISK (1471)	PATN5

Loc	Routine or Structure	Name
1352	New Page Request	NPAGE
1400	Get First Device	GTFDV
1427	Begins APU Code for Startup	DOSTART
1460	Drum Startup	STDRUM
1471	Disk Startup	STDISK
1500	Find Drum Page Table Entry	FDPTE
1512	Find Band in Free Drum Page Table	FBFDPTE
1523	Make Drum Address	MDRM
1531	Parse Drum Address	PDRM
1536	Clear/Set Drum Free Page Table	CSFDP
1546	Check for page in core routine	INCOR
1563	Execute command (used to set up for TSU commands to channel)	EXEC

6.3 Detailed Microcoded Routines Descriptions

The following pages contain descriptions of major routines which are microcoded. By referring to the description of the appropriate data structure, one should be readily able to read the microcode.

Control is retained in the main loop (microcoded) until there is something for the AMC to do, then control is passed to an APU routine. Each of these APU routines has a real time constraint of about one quarter millisecond in order to run the AMC at full speed.

6.3.1 Subroutine Linkage

Due to the complex nature of the swapper, several levels of subroutines were necessary. A stack mechanism was designed for saving the link. The stack begins in the highest numbered scratchpad register (77B). The subroutine affects three holding registers. STKP is reserved for the stack pointer. It is initialized to 77B in the main loop. The registers R5 and R4 are also used by the stack link (STKLK) subroutine. Other registers are preserved (although not without some effort).

6.3.2 Main Loop (MAIN) (Also described in Section 6.1)

Control remains in this loop as long as there is nothing for the AMC to do. While the AMC is idle, no memory queues are generated. The AMC has two types of requests which can originate from another processor. Each request is accompanied by a request strobe which sets the request latch in the AMC microprocessor.[1] The TSU requires periodic processing by the AMC. The TSU sends an ATTENTION signal every time it needs another instruction. The AMC then determines if there is an instruction in the holding register which has just been executed. If there is, it does the necessary cleanup for it. Then it attempts to find transfers for the devices. In addition, this loop calls two other APU-coded routines. One of these routines will copy entries from the disk cylinder queues to the disk sector queues incrementally. It will not execute longer than about 200 microseconds. The other routine will maintain the requests to bring a process into core. It is so located in the loop that it will be called about once a sector time (1 millisecond).

---

[1]See discussion of Requests in Section 4.2.

The main loop consists of two large routines and some small ones. The large routines are called MAIN and PATTN (process attention). We begin at MAIN:

a. Set the scratchpad stack pointer and the APU core stack pointer.

b. Process Attention by calling PATTN. This routine will test the attention signal and take the appropriate action. If no attention is set, it tests the request strobe latches and returns a value to signify that request latch 1 is set or reset.

c. If request latch 1 is not set and if either the number of entries on the disk sector queues equals zero or the flag register bit Ø is set, start the APU routine which will copy entries from the cylinder queues to the sector queue, then go to a.

d. If request latch is set, reset it.

e. If the activate cell is one, start the APU on the Activate Request routine by calling SAPU.[2] Punt if the Activate Queue is empty and the activate cell is one.

f. If the General Request queue is empty, go to a. Otherwise start the APU on the General Request routine by calling SAPU.

g. Process Attention by calling PATTN. (See b. above)

h. If either the number of entries on the disk sector queues equals zero or the flag register bit Ø is set, start the APU routine which will copy entries from the cylinder queues to the sector queue.

---

[2] See Start APU in Section 6.3.4.

i. Process Attention by calling PATTN. This routine returns a value which indicates that request latch 1 is set or reset. If set, go to d, otherwise go to f.

6.3.3 Process Attention (PATTN)

The idea of this routine is to determine whether any TSU needs servicing. As a convenience, when the attention latch is not set it returns a value dependent on Request latch 1. It will also save the state and wait until the Breakwait cell becomes non-zero if Request latch 2 is set.

If an attention latch is set, it begins by looking for a TSU which needs cleanup. The position counter has a bit specially designed to aid this search.[3] When a TSU is found needing cleanup the APU is given control. When the APU returns the search is begun again. When the search fails another search is initiated, this time for a TSU needing a transfer. Again there is a bit in the position counter (holding register awaiting execution) specially designed for this purpose. When a TSU is found needing an instruction (startup), another APU routine is begun. When the APU returns, the search for a TSU needing an instruction proceeds. When the search fails the attention latch is tested. If it is set, we begin both searches again. Otherwise an APU routine is called which decides whether another process is required. Then control is sent to the beginning of this routine.

a. Go to c if attention set (it is reset by the test).

---

[3]See TSU Position Counter discussion in Section 3.2.2.

b. Save the state and wait on Breakwait Cell if request latch 2 is set. Otherwise return -1 if no request latches set, otherwise return Ø.

c. Set TSU # to Ø.

d. Go to f if the TSU tested does not hold an instruction which has finished execution but has not been read by AMC.

e. Compute the clean-up buffer pointer and store it into the first word of the TSU state. Store the rest of the TSU state in core and then start the APU on the clean-up routine by calling SAPU (which will punt if the clean-up buffer is empty). Control is returned to c.

f. Add 2 to the TSU #, i.e., search only TSU Ø and 2. If the TSU # is less than 4, go to d. (Currently, only two TSUs exist; number Ø for the drums and number 2 for the disks)

g. Set TSU # to Ø.

h. Go to j if TSU tested does not need an instruction.

i. Set core stack and determine whether the disk or drum routine should be called. If an instruction is sent to the TSU, control is returned to g. Otherwise, control goes to j.

j. Add 2 to the TSU number. If the TSU number is less than 4, go to h.

k. If attention latch set, go to c (attention latch reset by test).

l. Call the APU routine which determines when to start bringing another process into core. Then go to a.

6.3.4 Start Auxiliary Processing Unit (SAPU)

This is a short routine which does some common things before going to the APU code. It takes an address to begin the APU and a pointer to a request. The routine fails if pointer to the request is an end of list (777777B) word.

a. Put address of APU routine in Program Counter.

b. Put address of node in B. Fail return if it is an end of list.

c. Fetch the request code from the request and put in A and X.

d. Stack the link and go to APU.

6.3.5 CHT HASH (CHTHSH)

This routine computes a pointer to a pseudo-node, the last entry of which is a pointer in CHT1. It takes the 48 bit Unique Name as input. These two words are EOR'ed together to form a single word. This word is then split into 3 separate 8-bit bytes. Each byte is EOR'ed together and the result EOR'ed with 264B. The resulting word is masked to leave the low order eight bits which are added to the appropriate base address.

6.3.6 CHT Search (CHTSCH)

This routine searches CHT for an entry which contains the Unique Name presented. This routine finds the first entry (if there is one) by calling the subroutine CHTHSH which computes the address of a pseudo-node in CHT1. The collision pointer[5] of this pseudo-node is a pointer (or end of list, 777777B) into CHT2. The collision pointer list is followed until the last entry is found or the Unique Name presented compares with the Unique Name

[5]The CHT data structure is discussed in Section 4.2.

in one of the entries. If the search succeeds, the routine returns with the address of the entry. Otherwise, it fail returns with a pointer to the last node on the collision chain.

If the Unique Name is zero, it is a failure, otherwise:

a. Find the pointer to the first pseudo-node in CHT1 by calling CHTHSH.

b. Check the pointer for being an end of list; if it is, fail return.

c. Compare the Unique Name in the entry with the one given. If they are equal, return.

d. Get the new pointer from the collision pointer field (word 5 of entry) and go to b.

6.3.7 Enter CHT Entry (ECHT)

This routine does all the chain patching such that the entry presented represents a page in core with the given Unique Name. A CHT entry (in CHT2) may only represent a page if it can be reached from an entry in CHT1 by following the collision pointer chain.[6] The entry must not be on another collision chain when presented to this routine. It is a fatal error to present a Unique Name which already exists as a page of memory, i.e., may be found by CHTSCH. The Unique Name and disk address are placed in the entry.

a. Pass the Unique Name to CHTSCH. If it succeeds in finding the Unique Name, PUNT.

b. Store Unique Name and disk address into node.

c. CHTSCH returned the pointer to the last entry on the collision pointer chain for the given Unique Name. Replace

[6]The CHT data structure is discussed in Section 4.2.

the collision pointer in this entry with a pointer to the new entry.

d. Put End of List (777777B) into the new entry's collision pointer field.

6.3.8 Delete CHT Entry (DCHT)

The effect of this subroutine is to remove the page with the given Unique Name from core. This is accomplished by removing the entry with the given Unique Name from the collision chain[7] which begins in CHT1 for the given Unique Name. If such an entry does not exist the subroutine fail returns. A pointer to the entry is returned if it is found and removed. Only the collision pointer field in the preceeding entry is affected.

a. Find the entry by calling CHTSCH. If the entry is not found, fail return.

b. Get the collision pointer from the node found. This pointer points to the next entry on the collision chain.

c. Put the pointer into the collision pointer field in the entry preceeding the entry found by CHTSCH. CHTSCH returns a pointer to the preceeding entry.

6.3.9 Clear CHT Entry (CCHTE)

The entry is cleared in all fields except for the page number which always remains the same, and the free core list pointer field which if non-zero indicates the entry is on the Free Core List. Prefetches are used to prepare memory so that this routine will execute in a minimum time.

---

[7] The CHT data structure is discussed in Section 4.2.

6.3.10 Get Free Core (GFC)

The goal of this routine is to return a free page of core by returning a pointer to a CHT entry. The entry returned will have been removed from any collision chain it may have been on, and cleared to zero. In the process of attempting to find a page which is free, it follows the free core list pointer chain, removing those entries which are not free from the Free Core List. The Free Core List is a standard circular list.[8] If no free entry can be found the routine fail returns.

a. Set protect 2 and initialize search by getting pointer to pseudo-node which points to first entry. Go to lower case.

b. Remove entry from Free Core List which is not free. Decrement number of entries on free core list counter (NFCL). Continue search at the entry pointed to by the one thus removed.

c. Get pointer to next node. If it is the End of List (777777B) unprotect 2 and fail return.

d. Test all pertinent fields for zero. If any of the fields are non-zero go to b. The fields tested are:

   1. Scheduled count

   2. Dirty Bit

   3. Disk Write Bit

   4. Bits 3-5 of the Page Status word

   5. Lock field

e. Remove the entry from the Free Core List. Decrement the number

---

[8] See description of standard circular list in Section 4.5.

of entries on the Free Core List. Remove the entry from the collision chain it may be on. Clear the maps of the CPUs if the Unique Name is not zero. Clear the CHT2 entry and clear the Free Core List Pointer field in the entry.

6.3.11 Put Page on Free Core List (PPFCL)

Given a pointer to a CHT entry, the entry will be placed on the Free Core List provided that the Free Core List pointer is zero. This proviso is necessary because a page may be used, written out, used again and written out again, all while the page is on the Free Core List. Whether the entry is put on the Free Core List or not, the routine returns successfully.

a. Return if the Free Core List pointer not equal to zero.
b. Increment the number of entries on the Free Core List.
c. Stack the entry on the Free Core List by calling the subroutine SEL.

6.3.12 DHT HASH (DHTHSH)

The purpose of this routine is to compute the index of the entry in DHT from the given disk address. DHT is organized as two tables (which must be adjacent and ordered DHT1, DHT2). The first table has one word per entry and the second two. The hash code (address) is computed as follows:

a. Zero the top two bits of the word given as the disk address.
b. Rotate the result so that the top half of the word is in the place of the bottom half.
c. EOR the rotated word with the original disk address.

d. Extract the bottom 13 bits of the address to form an index into DHT. The size of DHT is normally smaller than 8K, therefore perform a modulo operation.

e. Return index if index is less than size of DHT.

f. Set index to index minus size of DHT. Go to e.

6.3.13 Search DHT (DHTSCH)

The goal of this routine is to locate the presented disk address in the Drum Hash Table.[9] The Drum Hash Table is composed of two parts, DHT1 and DHT2. DHT1 has one word per entry (the disk address) and DHT2 has two. If during the search a zero is found instead of the presented disk address, the search fails. The search will also fail (after a long time) if the table is full. It will fail immediately if the presented address is zero.

a. Fail return if address equals zero.

b. Compute the index by Hashing the disk address. This amounts to a call on DHTHSH.

c. Set up pointers into the two tables. Set the top two bits of the disk address. This will allow the search to succeed independent of the top two bits in the DHT1 entry probed.

d. Fetch the first entry at the starting address.

e. Fail if the word fetched is zero.

f. Set the top two bits of the word fetched.

g. Return if the disk address just fetched matches the one prepared in c.

---

[9] See Drum Hash Table discussion in Section 4.3.

h. Increment the two pointers. If the pointers overflow the table, reset the pointers to the beginning of the table.

i. Increment the counter of the number of probes. If it exceeds the size of the table fail return.

j. Fetch the next word from the table and go to e.

6.3.14 Make DHT Entry (EDHT)

The object of this routine is to make a complete entry into DHT. It will fail if an entry with the same disk address is in DHT already. It will also fail if the disk address presented is zero.

a. Fail return if the disk address is zero.

b. Find the pointers into DHT by searching for the disk address presented. If the search succeeds we cannot make the entry, therefore fail return.

c. Copy the three word entry from scratchpad to the appropriate places in DHT.

6.3.15 Delete DHT Entry (DDHT)

Given a disk address, this routine will find it and delete it from the Drum Hash Table. However, it is not through yet, for the method of handling collisions is to scan linearly through the table for an empty spot. This implies more work needs to be done. Suppose disk addresses a, b, c all produce the same address with the hashing algorithm (see Fig. 6.2). Then they would occupy sequential locations in the table corresponding to the order in which

they were entered into the table. Suppose we delete the second entry (Fig. 6.2b). It is now impossible to find the third entry with the search mechanism described; the third entry is no longer correctly placed in the table. It is therefore necessary to re-enter all entries which occur between the

a
b
c
∅

Fig. 6.2a Three Hash Table Entries

a
∅
c
∅

Fig. 6.2b b is Deleted

a
c
∅

Fig. 6.2c Corrected Hash Table if a,b,c Hash into same Location

a
Ø
c
Ø

Fig. 6.2d Corrected Hash Table if c Hashes into Location it is in

deleted entry and the next free entry (Ø) (Fig. 6.2c). It is not sufficient to move them up one slot, for suppose only the first two hashed into the same address and the third hashed into the address in which it now resides (Fig. 6.2d).

a. Search for entry by calling DHTSCH. If search fails, fail return.

b. Delete entry, set up scan, go to f.

c. Return if entry deleted.

d. Copy entry into scratchpad, delete entry.

e. Reenter entry into DHT.

f. Fetch the first word of the next entry in DHT.

g. Increment the pointers into DHT. If the pointers overflow the table boundaries, reset them to the beginning of the table and fetch the first word of entry. Go to c.

6.3.16 Append Entry onto List (AEL)

This routine appends a node onto a list. It takes as arguments

pointers to the list header and the node and the offset of the list pointer in the node. It affects only the pointer fields. The second word of the list header points to the last node on the list. A pointer to the node to be appended is placed in the current last node and in the second word of the list header. The end of list mark (777777B) is placed in the new last node.

6.3.17 Stack Entry on List (SEL)

This routine puts the node presented onto the given list.[10] Offset must be given by the caller. The first entry of the header points to the beginning of the list. The second entry points to the end. The pointer in the node is located at the beginning of the node plus the offset. If the list is empty the node presented becomes the first and last node.

a. Fetch and save the first word of the header. Store the pointer to the node into the first word of the header.
b. If the first word was the end of list, store a pointer to the node into the second word of the header.
c. Store the pointer found in the first word of the header into the pointer field of the node.

6.3.18 Remove Entry from List (REL)

The purpose of this routine is to remove an entry from the list presented. The list structure has only one-way pointers, i.e. only one pointer per entry. Therefore this routine also requires a pointer to the entry preceeding the entry to be removed. If no entry follows the entry given the routine fails. This allows this routine to be very general. Specifically, another entry point is called Remove top entry. If it fails it implies that the list is empty. The

---

[10] See discussion of Standard List Structure in Section 4.5.

routine moves the list pointer from the node to be removed to the preceeding node and returns a pointer to the node removed. Only the pointer fields are affected.

a. Fetch the pointer node of the node preceeding the node to be removed.
b. Fail return if it is an end of list mark (777777B).
c. Fetch the pointer word of the node to be removed. If the pointer is an end of list mark it is the last node on the list. Therefore set the second header word with the pointer to the node preceeding the one to be removed.
d. Store the pointer obtained from the node to be removed into the pointer field of the preceeding node.

6.3.19 Save State (SAVST)

This routine is entered with M in a scratchpad register and RØ in M. It saves the state of all main registers, the holding registers, and all but one scratchpad register (the one holding the contents of M). As it stores the state it zeros the Breakwait Cell. When the state is stored it fetches Breakwait until it becomes non-zero. The state is then loaded. Every register is restored except the scratchpad register holding M and the effect of a DGOTO in the instruction preceeding the one where the break occurred (OREG).

6.3.20 Dump TSU State (DPTSU)

This subroutine requires the address of the place to store the state

and the TSU #. It stores the TSU #, then stores the state. The routine is three loops, for getting the position counter requires some extra logic which is in the routine GETPOS.

a. Store the TSU #.

b. Setup to get the first 9 registers. This loop slides a bit to the left once each time through the loop. The low order 16 bits in the select register are addresses of registers in the TSU.

c. Setup to get the position counters in the order Ø, 1, 2, 3. This loop uses a counter for the Unit # as required by the GETPOS routine.

d. Finally setup to get the last 3 registers using the sliding bit technique.

6.3.21 Generate Wakeup

This routine is the primary communication with the microscheduler. It takes a pointer to the process table and a data word. The microscheduler maintains a stack into which these two words may be put. The stack is full if the stack pointer (USIBTOP) plus 2 has Ø in the low order five bits.

a. Set protect $1Ø_8$.

b. Fetch USIBTOP and add 2 to it. If the result exceeds the top of the stack (five low order bits are zero) unprotect $1Ø_8$ and go to a.

c. Store the new value into USIBTOP.

d. Store the pointer to the process table merged with the wakeup command into the word pointed to by USIBTOP. Store the data word into the next word.

e. Unprotect $1Ø_8$.

6.3.22 Send TSU Instruction (STSUI)

The purpose of this subroutine is to do all the work of sending a complete list of TSU instructions.[11] It also returns to the next highest level on the stack. Several arguments are expected to be in the Main and Holding registers of the microprocessor. Only the Z register is changed by this routine.

a. Send instruction from R1.

b. Send device address from R2.

c. Send page and map address from R3.

d. Send word count from Z (effectively).

e. Send Unique Name word Ø from M.

f. Send Unique Name word 1 from Q.

g. Send Unit number from R1.

h. Return to next higher level on stack.

6.3.23 Stack Entry on Free List (SETFL)

This function keeps track of unused request entries.[12] When the entry is no longer useful to the AMC, it must be explicitly placed on the Free Request Entry List by calling this routine. It must not be put on by merely appending it to FREL. The CPU uses the free requests on this list, so a protect must be set before the entry can be appended. Furthermore a count of the number of entries is maintained by this routine.

a. Protect 2.

b. Increment NFREL (Number of free requests).

---

[11] See discussion of select register in Section 3.2.1.

c. Stack entry on Free List.

d. Unprotect 2.

6.3.24 Remove Entry from Free List (REFFL)

If possible, this routine returns a pointer to a Free Entry.[13] If no entry exists, a counter (ROFN) is incremented and the routine fail returns. Free Entries contain six words. They are kept on a standard circular list whose header is called the Free Request Entry List (FREL).

a. Protect 2.

b. Remove the top entry from Free Request Entry List.
If no entry exists go to e.

c. Decrement number of free requests (NFREL).

d. Unprotect 2 and return.

e. Increment ROFN.

f. Unprotect 2 and fail return.

6.3.25 Initialization Sequence for AMC

When the AMC microprocessor executes the instruction at zero, it makes the choice of saving the state and waiting for a breakpoint, or initializing enough core for the system to get started. The initialization sequence begins in instruction 1. One cell in memory called the Switch Register in Memory (SRMEM)[14] contains a few bits which this routine interprets to obtain the following information:

a. Whether the AMC is to fill core from a device.

b. Whether the AMC is to save core.

[12, 13] See discussion of Request Entries in Section 4.2.

[14] See description of Crash and System Areas on Disk and Drum in Section 3.3.

c. The type of device (i.e., drum or disk) on which the system will be found.

d. The unit number of the device.

e. The addresses of the place to put the saved core (i.e., old system) and the place to get the new system.

Sixty four thousand words are saved and read into core. There are four possible sets of addresses on the device selected. Eight devices may be selected (one from each TSU, either unit Ø of 1).

a. Reset both request strobe latches.

b. Hang until a request strobe is received.

c. Reset request latch 1.

d. If the CHIO is doing the Initialization, go to q.

e. Set up loop for drum or disk depending on TSU field in SRMEM.

f. Set up unit number.

g. Wait until device has become idle by calling RDST.

h. Go to 1 if not supposed to store old system.

i. Send commands to TSU by calling STSUI. The registers are not changed by that routine.

j. Wait for drum to idle by calling RDST.

k. Check for any errors in transfer. If any errors occurred, go to h.

l. Update addresses in holding register. In updating the device address, if it is found to be on the last sector (depends on the device), add an appropriate value which will add one to the band and zero the sector field.

m. Increment a down-counter. If it is not zero, go to h.

n. Compute the address from which the system will be read. This involves zeroing the sector field and adding one to the band field of the device address.

o. If the instruction sent to the TSU is not a read, go to h.

p. Send request strobe 1 to microscheduler.

q. Wait for request latch 1 to be set.

r. Reset request strobe latch 1 and load state by branching to LOADST.

6.3.26 Wait Until Device Idle (RDST)

This routine is used to wait until the selected device is idle. It reads the position counter and keeps reading it until certain conditions are true.

a. Read position counter zero of selected position counter.

b. Go to a if the position is not valid.

c. Return if all of the following conditions are true:

    1) There is no instruction awaiting execution in the holding registers.

    2) There is no instruction being executed in the functional register.

    3) The position of unit Ø is in the gap or first third of <u>any</u> record (512 word).

6.3.27 Compute Next Sector on Selected Unit (FSTR)

It is necessary for the AMC to know the next sector on any unit at any time. While a transfer is taking place, the holding registers must be loaded for the next transfer. Since there is more than one unit attached to one TSU we must be able to determine what sector of the desired unit will follow the sector currently being processed by the functional side of the TSU.

The routine may fail to produce a sector for one of two reasons. It may be that there is less than one record time until the swapping of the registers (256 μsec). This would not give the AMC enough time to select the proper transfer. It may be that the TSU contains an instruction, but is delaying its execution for a period exceeding one sector time (1 millisec). In this case we do not wish to load the holding registers since the total situation may change by the time the instruction is executed.

a. Read SP, the position of the selected unit, by calling GETPOS. See Fig. 6.3a for the format of a position.

b. Read FP, the position of the functional unit, by reading the functional unit number and calling GETPOS.

c. Fail return if the functional unit contains an instruction but is delaying execution, i.e., the registers will not swap at the next end of sector.

d. Fail return if the remaining time before the swapping of the registers is less than or equal to one record. This insures that we have 250 μs to figure out what to do before the registers swap.

e. Compute the distance to the swapping of the registers. This is

D=20B-(FP Λ 17B), since the registers swap at the end of the sector and FP Λ 17B gives the position within the current sector. Add D-1 to SP. Now SP tells us where the selected unit will be just before the registers swap. We are interested in the first sector after the registers swap. We get this by adding one sector (20B) more to SP, which now contains an address somewhere in the first sector which will come up after the registers swap, except for

f. the fact that a comparison of positions for two different units may be off by 2. To correct for this (conservatively) add 2 to SP if the selected unit differs from the functional unit.

g. Now SP points somewhere into the sector we want. We extract the address of the sector, discarding irrelevant record and PWR bits, and take it modulo the size of the unit (24 for drum, 5 for disk).

6.3.28 Get Position of Rotating Device (GETPOS)

This routine will return the valid position register for the TSU and unit selected. It must compute from the unit number the bit to set in the Select Word of the TSU.[15] It does this by shifting a bit appropriately. If the position returned by the TSU is not valid, the TSU is asked again.

a. Compute select word

b. Request TSU to send position counter selected

c. Return if position valid, go to b if not.

---

[15] See discussion of Select Register of TSU in Section 3.2.9.

7. MEMORY MANAGER SOFTWARE

This section describes the software portion of the memory manager.

7.1 APU Code - Overview

As shown in the main loop, (Figure 6.1) there are dispatches to the APU code at various points. Basically, the APU code routines handle five categories of tasks:

1) General requests
2) Activate requests
3) Cleanups
4) Startups
5) Miscellaneous

General requests and activate requests were described in Sections 4.1.1 and 4.1.2 respectively. Using the request code, one dispatches via a transfer vector to the specific APU routine that handles that case and then returns to the main loop.

Cleanups and startups were discussed in Section 6.1. A cleanup checks to see if the TSU properly executed the last command and handles possible errors, while a startup prepares and then sends commands to the TSU. As with general and activate requests, there are different cases for startups and cleanups. The request code field of the node is used as an index into transfer vectors to the correct startup or cleanup routine.

The miscellaneous category includes other things shown in the main loop such as reading in context blocks, reading in process core working set pages, and several other things the memory manager does. These are invoked through a primary (main) transfer vector.

The dispatching structure is shown in Fig. 7.1.

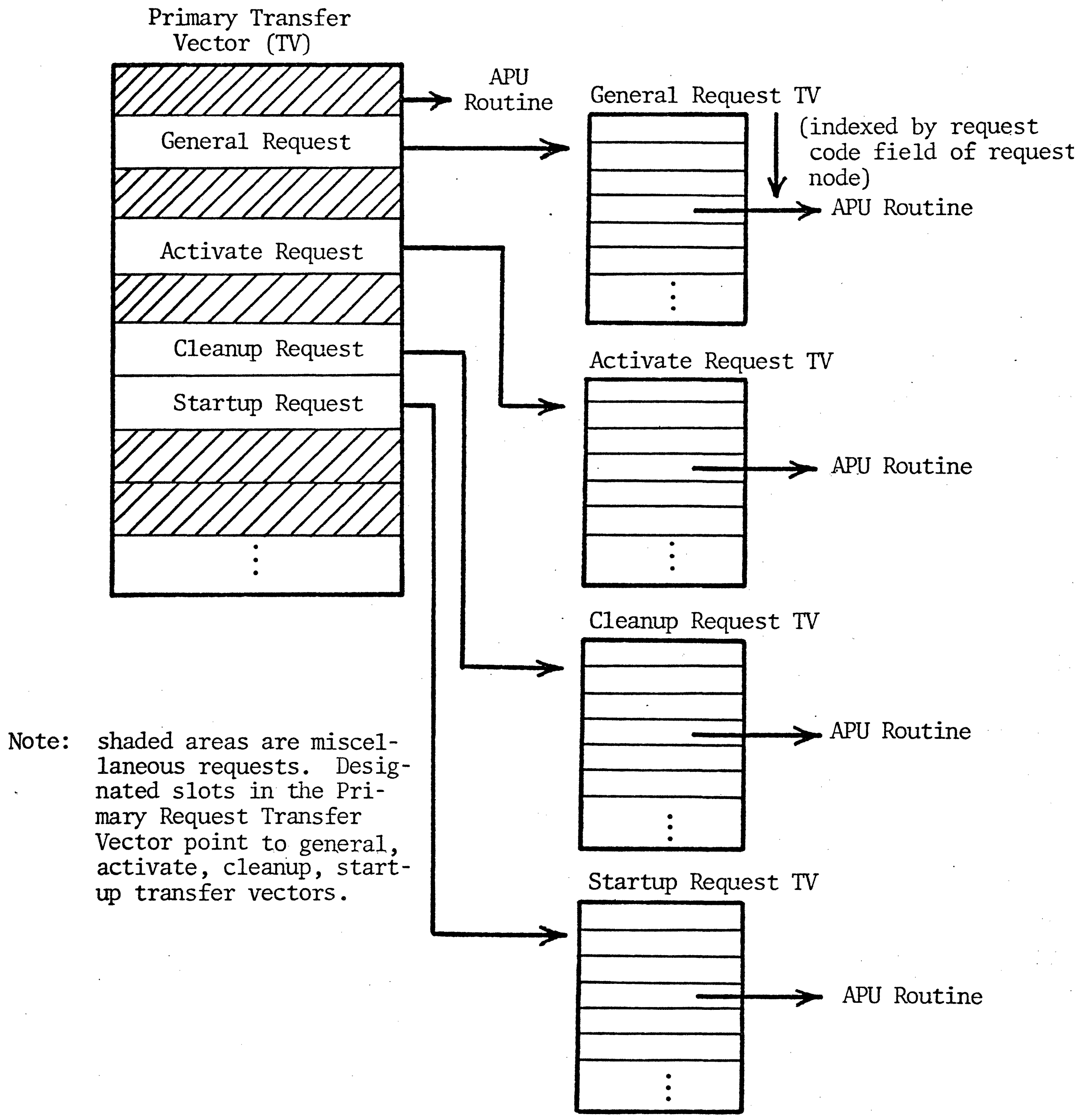
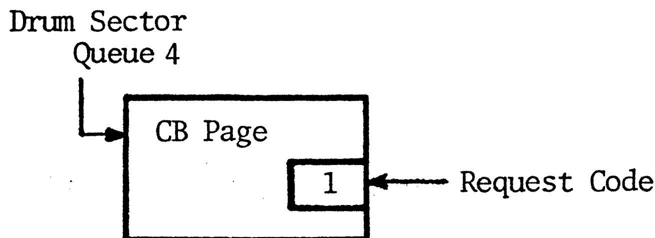


Fig. 7.1 Dispatching Structure

We have already seen how a process is loaded in the section on queues (Section 4.1) but it was from a perspective of queue use. We shall now use the same sample but view it from a different vantage point, that of seeing how different startups and cleanups are invoked.

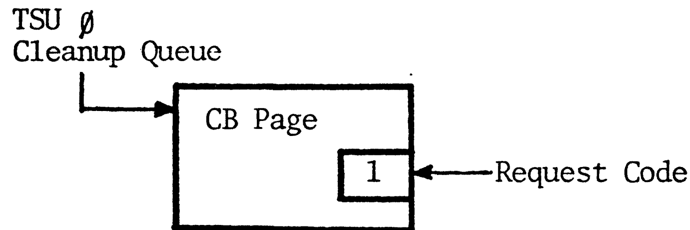
The sequence starts when the memory manager has time to read in a process' context block. It dispatches via the primary transfer vector to the APU routine that handles context block reads (miscellaneous request). This APU routine finds out where the context block is (for example, drum sector 4), and puts a context block read request on drum sector queue 4. (If one is not familiar with Section 4, Section 4.1 should be read before proceeding.)



The request code value "1" denotes "context block read." This completes the APU routine task.

Later, while the heads are over drum sector 3, the memory manager does a startup for drum sector 4, to setup the TSU holding registers prior to coming to sector 4. It indexes into the startup transfer vector using the request code which in this case is 1. Thus in this example the APU routine corresponding to an index of 1 in the startup transfer vector is the startup for context block read. The command is sent over to the TSU

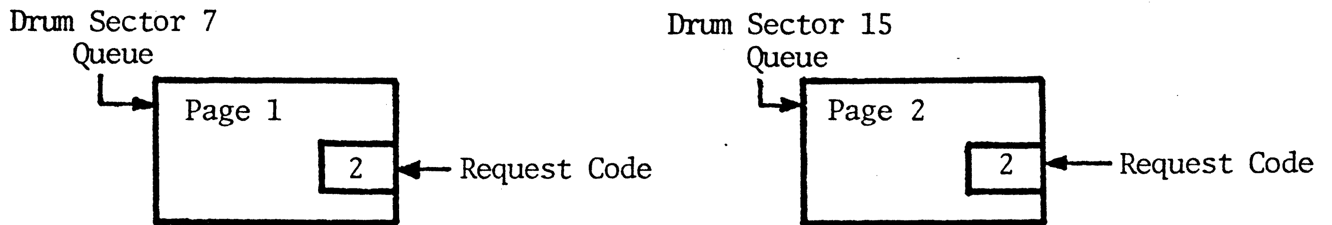
(TSU Ø) for the drum. Also, the request node is put on the cleanup queue for the TSU. The routine then returns to the main loop.



After drum sector 4 has passed, the memory manager responds to the ATTENTION delivered by the TSU by performing a cleanup. Similar to startup, it indexes into the cleanup transfer vector using the request code. The APU routine corresponding to an index of 1 in the cleanup for the context block read. As part of cleanup, a request node is put on the context block queue (CBQ). This is a queue of context blocks that have been read in, but need to be scanned to queue up page reads for the core working sets of these context blocks.

When the memory manager has time to read this CBQ, and upon finding the entry for the context block just read, it dispatches via the primary transfer vector to the APU routine that queues up reads for the different pages belonging to the core working set of the process. (This is a miscellaneous request.) Suppose there were (for simplicity's sake) only two pages to be read in, on drum sector 7 and drum sector 15.

Thus we have:



Note that the request code in the nodes have a different number from context block read.

In the same manner in which we did startup and cleanup for context block read, we do it for these pages at the appropriate time. The difference is that the APU routines dispatched to correspond to indexes of 2 in the startup and cleanup transfer vectors.

The reading of the last page (page 2 in this case) of the process core working set means that the manager has completed reading in the process working set. It then notifies the scheduler.

7.2 Major Transfer Vectors

The following are the five major transfer vectors found in the memory manager.

7.2.1 Primary Transfer Vector

Primary Transfer Vector	TV Index/Request Code
PUNT	0
General Request	1 → General Request Transfer Vector
Activate Request	2 → Activate Request Transfer Vector
Read Drum	3
Startup New Page	4
Cleanup	5 → Cleanup Transfer Vector
Queue Process Pages	6
Read Context Block	7
Copy from Disk Cylinder to Sectors	8
Startup	9 → Startup Transfer Vector
Write Page Startup	10

7.2.2 General Request Transfer Vector

General Request Transfer Vector	TV Index/Request Code
Remove General Request	0
Write Process onto Dram	1
Remove General Request	2
Write Process onto Dram	3
Direct Drum Transfer	4
Direct Disk Transfer	5
Return Page to Drum	6

7.2.3 Activate Request Transfer Vector

	TV Index/Request Code
Reserve Page Request	0
Release Page to Drum	1
Drum → Disk Transfer	2
Write Unique Name	3
Disk → Drum Transfer	4
Get free page	5
Destroy page	6
PUNT	7
PUNT	8
Read Page (for diagnostic purposes)	9

7.2.4 Cleanup Transfer Vector

Cleanup Transfer Vector	TV Index/Request Code
PUNT	0
Drum Cleanup for CB read	1
Drum Cleanup for process pages read	2
Drum → Disk transfer (drum cleanup)	3
Drum Cleanup for direct I/0	4
Disk Cleanup for direct I/0	5
Drum Write Cleanup	6
Drum Write Cleanup	7
Drum Cleanup for Destroy page	8
New page cleanup	9
Drum → Disk transfer (disk cleanup), write page	10
Drum → Disk transfer (disk cleanup), UN=0 check	11
Write Unique Name (disk cleanup), UN check for 0	12
Drum → Disk transfer (disk cleanup)	13
Write Unique Name (disk cleanup)	14
Disk Cleanup for destroy page (UN match check)	15
Disk Cleanup for destroy page (no UN match check)	16
Disk → Drum Transfer (drum cleanup)	17
Cleanup for read page diagnostic	18
Cleanup for disk cylinder seek	19

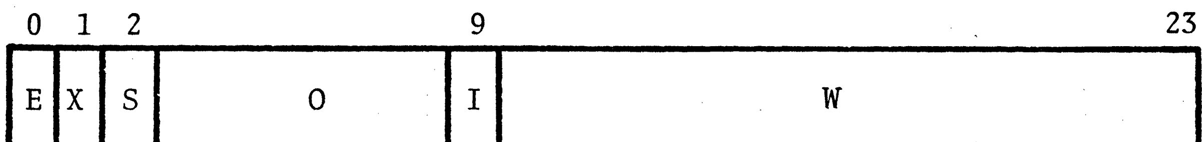
7.2.5 Startup Transfer Vector

Startup Transfer Vector	TV Index/Request Code
PUNT	0
Context Block Read	1
Process Pages Read	2
Drum → Disk Transfer (drum)	3
Direct Drum Transfer	4
Direct Disk Transfer	5
PUNT	6
PUNT	7
Destroy Page (drum)	8
PUNT	9
Drum → Disk Transfer (disk)	10
Check UN	11
Check UN	12
Disk → Drum Transfer (disk)	13
Disk Write Unique Name	14
Check UN	15
Destroy page (disk)	16
Disk → Drum Transfer	17
Read Page	18

7.3 The Auxiliary Processing Unit (APU)

The following is a description of the APU part of the memory manager. The microcoded main loop calls APU coded subroutines to perform the necessary processing. Conventions are made so that APU routines may call microcoded subroutines.

An APU instruction has the format



where:

E = ENVIRONMENT

X = INDEXING

S = Scratchpad - Memory operation

O = Opcode or scratchpad address

I = Indirection or Load/Store

W = Address

and if

E = Ø, effective address local

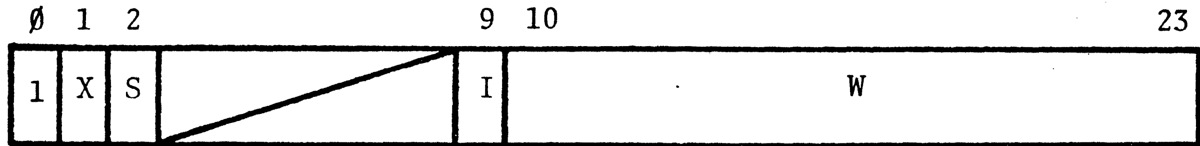
E = 1, effective address absolute

S = Ø, O is opcode, I is indirection

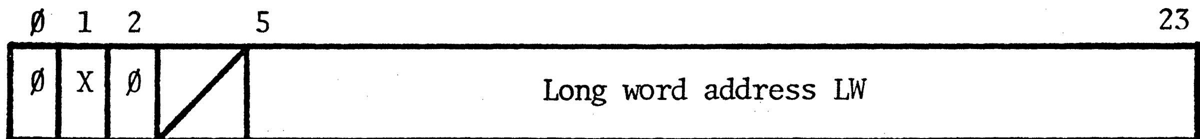
S = 1, O is a scratchpad address,
I determines Load/Store,

Indirection

There are two Indirect Address Words (IAW). A local IAW

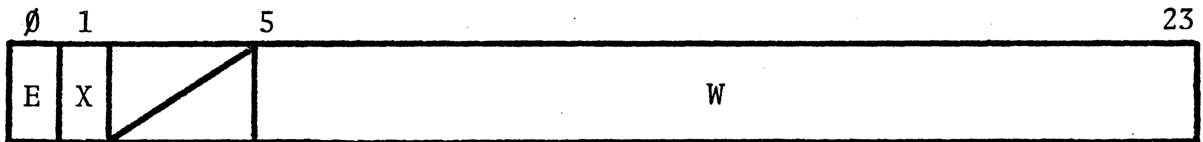


An absolute IAW:



Field Pointer

A field pointer is placed in B whenever a field operation (LDF,STF, ADF, MGF,ZRF) is to be done. It has the following format:



E = ENVIRONMENT

X = Indexing

IF E = Ø, W is an absolute address

E = 1, W is relative

X = 1, Q ← W+X mod 2^{18}

X = Ø, Q ← W

Registers

There are three central registers A, B, X, a 14 bit program counter P, memory relocation and bounds registers REL and BOUND, and a control stack CSTK. In addition all scratchpad registers are directly addressable

using either an RCH-like instruction (see below), or directly from core by setting S.

All of these registers are maintained in scratchpads. Before executing APU instructions the Main loop of the swapper initializes the CSTK to REL + 100B. It is advisable to avoid changing P, REL, BOUND, and CSTK with scratchpad instructions.

Addressing

Addressing is the same in all instructions except for Load and Store scratchpad. In the single exception there is no indirection. Indexing is modulo 2^{18} for all instructions. The effective address is computed as follows:

```
        Let I be an instruction,
            E(I) be the environment bit,
            IX(I) the index bit,
            IN(I) the indirect bit,
            Q the absolute effective address,
            QL the relative effective address, and
            X the index register.
        T ← I;
        Q ← I AND 37777B;
LOOP:   Q ← (Q + X) AND 777777B IF IX(T) ≠ Ø;
        QL ← Q;
        Q ← Q + REL IF E(T) = Ø;
```

*Indirection

```
IF IN(T) ≠ Ø DO;
    T ← CONTENTS(Q)
    (T ← T AND 37777777B & Q ← T AND 37777B &
    GOTO LOOP) IF E(T) ≠ Ø;
```

*Absolute Indirection

```
    Q ← T;
    Q ← (Q + X) AND 777777B IF IX(T) ≠ Ø;
    QL ← Q;
ENDF;
```

Instruction Set

This section describes all instructions. The symbols A, B, X, P stand for registers, O stands for opcode field, Q for the absolute effective address, QL for the relative effective address. $L indicates the contents of L on the right hand side of a left arrow and $L indicates a store into location pointed to by L on the left hand side of a left arrow. Certain fields within QL will be defined when appropriate. They will be used by saying F(L) which is to be interpreted as Field F (right adjusted) of L. The notation L_{SK} is to be interpreted as one symbol whose value is a scratchpad address.

Data Transfer:

```
LDA:  A ← $Q;
LDB:  B ← $Q;
LDX:  X ← $Q;
EAX:  X ← (X AND NOT 777777B) OR (QL AND 777777B);
```

LSK: ← (S = 1, I = Ø) \$O ← \$Q;

BEWARE: LSK MAY OCCUR IN AN IAW

STA: \$Q ← A;

STB: \$Q ← B;

STX: \$Q ← X;

XMA: T ← \$Q; \$Q ← A; A ← T;

SSK: (S = 1, I = 1) \$Q ← $\$O_{SK}$;

BEWARE: SSK MAY OCCUR IN AN IAW

LDI: A ← QL;

A ← A OR 7774B4 IF (QL AND 2ØØØØB) = Ø;

Note: allows loading of constants between

-2ØØØØB and 17777B;

Arithmetic Operations:

ADD: A ← A + \$Q;

ADM: \$Q ← \$Q + A;

SUB: A ← A - \$Q;

MUL: A ← A * (\$Q AND 7777B);

Note: a result $>2^{24}$ is not modulo 2^{24}, but is funny

MIN: \$Q ← \$Q + 1;

MDC: \$Q ← \$Q - 1;

ISK: SKØ(QL) a six bit scratchpad address

$\$SKØ(QL)_{SK}$ ← $\$SKØ(QL)_{SK}$ + 1;

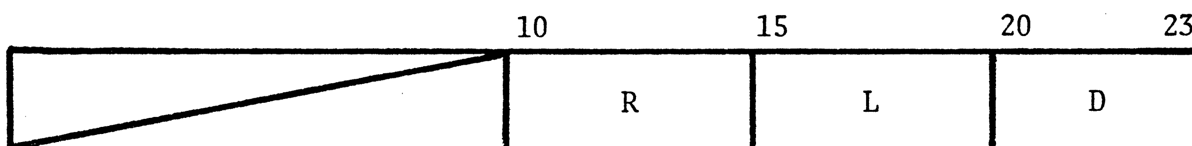
DSK: $\$SKØ(QL)_{SK}$ ← $\$SKØ(QL)_{SK}$ - 1;

Logical:

```
ETR:   A ← A AND $Q;
MRG:   A ← A OR $Q;
EOR:   A ← A EOR $Q;
```

Field Operation:

B contains a field pointer, QL is a field descriptor as follows:



where:

L: is the left bit (numbered from left to right)

R: is the right bit

D: is the displacement

The address of the field word is computed from B, X, and D:

```
FADR ← B;
FADR ← FADR + REL IF E(B) = 1;
FADR ← FADR + X IF IX(B) = 1;
FADR ← FADR AND 777777B;
FADR ← FADR + D;
```

(unreasonable as this seems)

The following field operations are defined (let FD be the specified field, i.e., bits L through R of $FADR):

LDF - Load field: loads the specified field right adjusted into A. All other bits in A are cleared.

```
A ← FD
```

STF - Store field: takes the number of bits specified by the field descriptor out of the least significant portion of A and stores them into the field.

FD ← A

ADF - Add to field: adds the contents of A to the specified field. A is unchanged.

FD ← A + FD

MGF - Merge field: Merges the (R - L + 1) low order bits of A into the specified field. A is unchanged.

FD ← A OR FD

ZRF - Zero field: zeroes the bits in the field as specified in the (R - L + 1) low order bits of A.

FD ← FD AND NOT A

Conditionals:

SKE: P ← P + 2 IF A = $Q;

SKNE: P ← P + 2 IF A ≠ $Q;

SKG: P ← P + 2 IF A > $Q;

SKGE: P ← P + 2 IF A >= $Q;

SKL: P ← P + 2 IF A < $Q;

SKLE: P ← P + 2 IF A <= $Q;

SKA: P ← P + 2 IF A AND $Q = Ø;

SKNA: P ← P + 2 IF A AND $Q ≠ Ø;

SKB: P ← P + 2 IF B AND $Q = Ø;

SKNB: P ← P + 2 IF B AND $Q ≠ Ø;

SKM: P ← P + 2 IF A AND B = $Q AND B;

```
SKUM:  P ← P + 2 IF A AND B ≠ $Q AND B;
SKR:   $Q ← $Q -1;
       P ← P + 2 IF $Q < ∅;
SKI:   $Q ← $Q + 1;
       P ← P + 2 IF $Q < ∅;
SKN:   P ← P + 2 IF $Q < ∅;
SKP:   P ← P + 2 IF $Q >=∅;
SKEL:  P ← P + 2 IF ($Q OR 77B6) = -1
SNEL:  P ← P + 2 IF ($Q OR 77B6) ≠ -1;
BRQ:   CALL R2PNT IF Request 2 set;
       P ← QL IF Request 1 set;
BRX:   X ← X + 1; P ← QL IF X < ∅;
BRPX:  X ← X + 1; P ← QL IF X >= ∅;
```

Unconditional branches:

```
BRU:   P ← QL;
BRM:   $Q ← P; P ← Q + 1;
BRR:   P ← $QL + 1 mod 2^14
BSR:   P ← $QL + 2 mod 2^14
BSL:   $CSTK ← P; CSTK ← CSTK + 1; P ← QL;
BSX:   X ← P; BSL;
BVR:   CSTK ← CSTK -1;
       P ← $CSTK + QL mod 2^14;
```

CALL: The intent of this instruction is to call microcoded subroutines. These may have a fail return, in which case no skip indicates failure and skip indicates success. The

register loaded with the APU fail address is given by bits 1Ø-12 of QL as follows:

Ø - No fail return (no skip from subroutine)

1 - Fail return loaded already

2 - Z ← fail return

3 - Q ← fail return

4 - M ← fail return

5 - R5 ← fail return

6 - FAØ ← fail return

7 - not defined

The state of M, Q, Z and RØ-R5 is maintained by this instruction in scratchpad. The state is loaded from scratchpad before executing the subroutine and returned to scratchpad when the subroutine has concluded. The correspondence is given by the following table:

M = A(SK5)	R1 = SK9
Q = B(SK6)	R2 = SK1Ø
Z = X(SK7)	R3 = SK11
RØ = SK8	R4 = SK12
	R5 = SK13

When the state is loaded, the microcoded subroutine at QL mod 2^{11} is executed.

```
SCALL: ($STKP)SK ← (NOT P) OR 7774B4;
       STKP ← STKP -1;
       CALL;
GOTO:  OREG ← QL
```

I/O To Devices on E Bus:

```
PIN:   Z ← A, Alert;
       A ← E1, PINSC;
POT:   Z ← A, Alert;
       Z ← B, POTSC;
```

Cycle:

```
LCY:   A ← A LCY QL;
```

Locate Leading one:

LLO: Left cycles the contents of A until

1) a one(1) appears in bit position zero(Ø) or
2) the shift count goes to zero.

The shift count is QL.

The index register is set to the shift count minus the bit position (the sign bit is bit position Ø).

Special functions:

```
STROBE:QL, Strobe;
UNPRO: QL; UNPRO;
PRO:   QL, PRO, RETURN IF PROTECT SUCCEEDS;
       QL, UNPRO, GOTO *-1;
CLRMAP: CLEARMAP;
```

Scratchpad to Scratchpad Operations

Let SKØ(QL) be one address, and

SK1(QL) be another

CSS: $\$SKØ(QL)_{SK} \leftarrow \$SK1(QL)_{SK}$;

CLS: $\$SKØ(QL)_{SK} \leftarrow Ø$;

CSSI: $\$SKØ(QL)_{SK} \leftarrow \$SK1(QL)_{SK} +1$;

CSSD: $\$SKØ(QL)_{SK} \leftarrow \$SK1(QL)_{SK} -1$;

CSA: $A \leftarrow \$SKØ(QL)_{SK}$

CSB: $B \leftarrow \$SKØ(QL)_{SK}$

CSX: $X \leftarrow \$SKØ(QL)_{SK}$

CAS: $\$SKØ(QL)_{SK} \leftarrow A$;

CBS: $\$SKØ(QL)_{SK} \leftarrow B$;

CXS: $\$SKØ(QL)_{SK} \leftarrow X$;

Register to Register Operations:

CLA: A ← Ø;

CLB: B ← Ø;

CLX: X ← Ø;

CLAB: A ← Ø; B ← Ø;

CLEAR: A ← Ø; B ← Ø; X ← Ø;

CAB: B ← A;

CBA: A ← B;

XAB: T ← B; B ← A; A ← T;

CBX: X ← B;

CXB: B ← X;

XXB: T ← B; B ← X; X ← T;

CXA: A ← X;

CAX: X ← A;

XXA: T ← A; A ← X; X ← T;

CNA: A ← -A;

CNX: X ← -X;

8. COMMUNICATIONS AND ERROR HANDLING

This section discusses the communications and the error handling aspects of the memory manager.

8.1 Communications - The Memory Manager and the "Outside" World

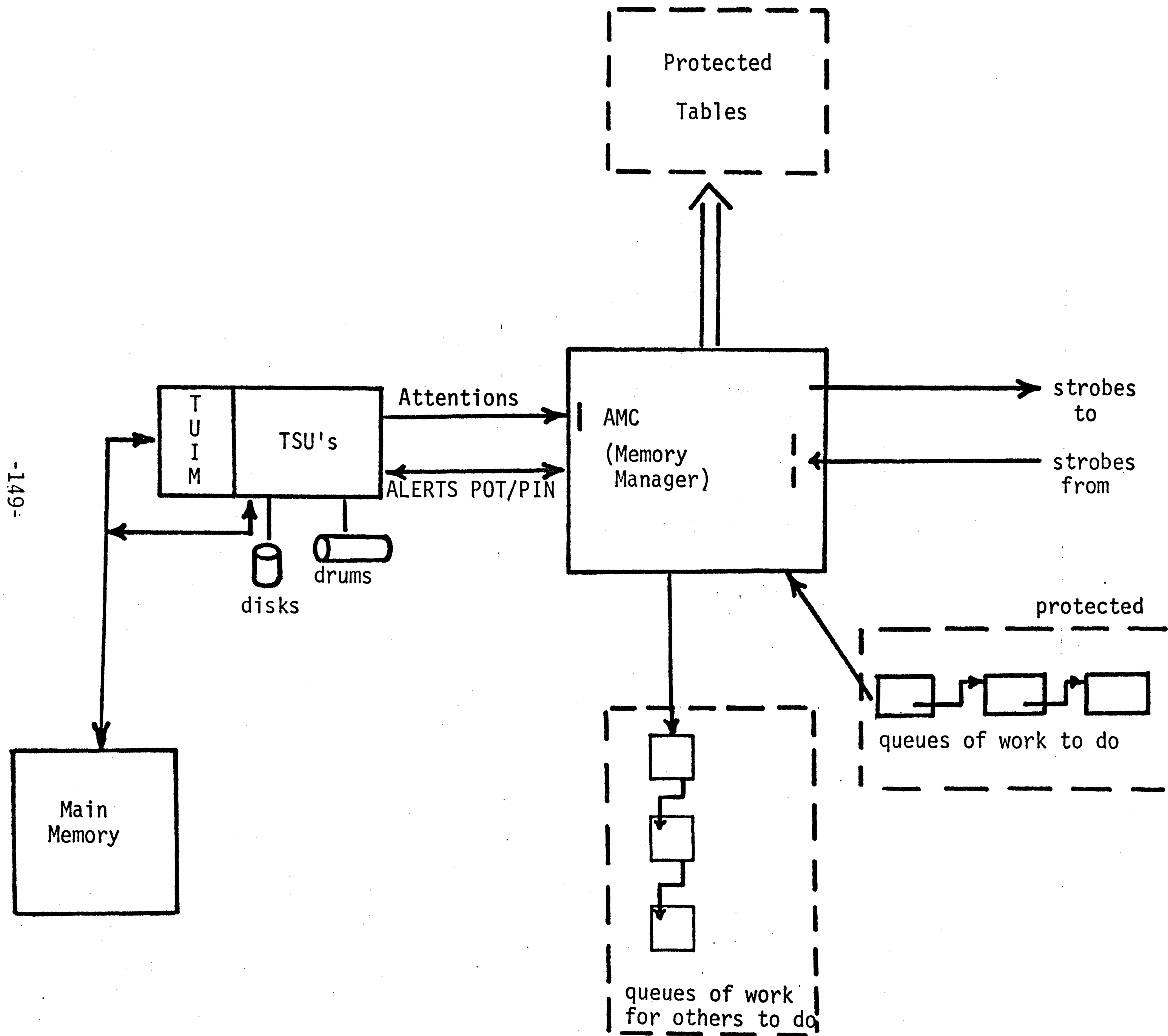
Up to this point, we have concentrated primarily on the memory manager and emphasized its ability to perform numerous tasks. We turn now to the conventions that exist between it and other components of the operating system.

Figure 8.1 illustrates the communications of the memory manager. Notice the use of queues, tables, and strobes. The strobe signals turn on latches (flags) in a processor. It is in this manner that the memory manager is notified that there is an activate or general request for it to service. Notice that it can strobe other processors as well.

The queues and tables are used by several processors, often to pass information between each other. There is a hardware processor interlock system for these data structures. When a processor is modifying one of these structures, it turns on the PROTECT associated with that structure. Other processors query the PROTECT and avoid handling the structure on finding the PROTECT set. Since PROTECTS are implemented in hardware, there is no race condition as occurs in setting software flags.

Different protects are assigned (by convention) to different structures. Thus a simple, convenient, but nonetheless powerful means of communication exists between the memory manager and other processors. Tables that are

Figure 8.1 Memory Manager Communications



handled include the CHT (core hash table) which has entries for every page in core. Both the AMC and the CPU handle CHT entries. The PRT (process table) which contains information relevant to each process in the system, is similarly protected. Various queues are used by processors to pass assignments to each other. One such queue is the general request queue into which other processors put nodes requesting work by the AMC. Such queues also are covered by the protection mechanism.

The communications between the memory manager (AMC) and controller (AMTU) has already been discussed at length in the section on hardware. (Section 3.1, 3.2.9). Actually the controller (AMTU) is to be considered an "insider" with respect to its close relationship to the memory manager.

Thus, the communication of the AMC is conceptually really quite simple yet affords the necessary communications capability of a multiprocessing environment.

8.2 Error Handling

8.2.1 General Error Philosophy

Insofar as possible, hard read errors are left to be handled by the process which requested the data. Write errors are not considered to be the responsibility of the process which initiated the write, and are handled more-or-less automatically by the MMS. It is assumed that hard errors, i.e. those that cannot be corrected by a reasonable number of retries, are infrequent and that clumsy and expensive ways of handling them

are therefore acceptable, although not actively sought after. The data is therefore discarded in this case. It may be retrieved using a direct I/O request.

8.2.2 Types of Errors

By 'error' we mean some fault in the process of reading or writing a page which is detected by the transfer hardware, or by code in the MMS which is very close to the hardware. We distinguish the following types:

1) UN errors, which occur when the unique name recorded on drum or disk does not agree with the one provided by the transfer request. All transfers read the UN and check it before transferring any data with the exception of drum writes. The check is for Ø rather than equality in the case of a disk write when the unique name is written. A normal disk write does not write the unique name.

Since the UN is not included in the checksum generated by the hardware, checksum errors have nothing to do with UN errors, and a UN error therefore always takes precedence.

2) Rate errors, (DATA TRANSFER LATE) occur when the memory cannot provide or accept data fast enough to keep up with the rotating device. A transfer which results in a rate error is always retried.

3) Memory system parity errors, other than parity on the data in core, are treated like rate errors.

4) A checksum error on data from the rotating device is called a soft read error. The transfer is retried until some number of soft errors have occurred. It then becomes a hard read error.

Only UN errors and hard read errors will be discussed in the rest of this section.

8.2.3 Errors During Swapping

These errors cannot occur during a write.

Read errors differ depending on whether it is the context block or a data page which is involved.

A UN or hard error on CB read is passed on to a special process which is responsible for such things, since the process is obviously unable to help itself.

8.2.4 Disk Read Errors

UN and hard read errors are handled in exactly the same way: they are reported to the process which requested the transfer (with a bit indicating which kind of error occurred). No DHT entry is made and any data read is discarded.

8.2.5 Disk Write Errors

There is no such thing as a soft write error.

A UN error on writing is reported to a special process, never to the process which initiated it. These errors are of three varieties:

1) UN on disk ≠ UN in request, for simple write request

2) UN on disk ≠ 0, for write request with WUN

3) UN on disk ≠ UN in request, for destroy request

The DHT entry, if any, is left untouched. These errors are always MMS failures, since the UN in the request is always the one on the drum.

8.2.6 Drum Errors for Non-Swapping Transfers

Drum errors can occur only for reads. There is only one kind of non-swapping read: a drum to disk transfer request. A hard read error is reported to the special process. This is disastrous, but it is rather unclear what to do.

9. CONCLUDING REMARKS

The preceding sections detailed the implementation of the memory manager. The major areas observed were hardware, microcode, software, system structures, data structures, and communications. Each had its role in fulfilling the goals as stated in the strategy.

The memory management system is conceptually quite straightforward. The hardware provides more information and capabilities than generally found. The microcode provides speed and efficiency. The APU code (software) is partitioned into discrete functions accessed by transfer vectors. The system structures and data structures are simple and few in type, while communications are quite clear. However, this organization is not typically found in systems, especially the incorporation of memory management into a separate micro-coded processor. Thus its very "uniqueness" may make it appear complex.

The system is currently running quite well. Unfortunately, we do not have quite as many drums and disks as were originally planned. We have two drums and two disks, which allows less swapping.

The concept of memory management being assigned to a separate processor has been shown to be viable. A fuller understanding of what constitutes memory management functions has resulted. Some of the shortcomings of the memory manager resulted from not having a precedent to follow in designing the memory management functions.

It is doubted that the present memory manager would be able to fully handle all the swapping that was intended. Having to read context blocks and queue up a lot of pages causes hiccups in the swapping. Activates have a similar effect. Possible alternatives are to make the basic memory

manager concerned with only startups and cleanups or put the swapper in yet another processor. Also, a lot more APU code could be put into microcode.

A problem with the microcode was that it was read-only, rather than read/write. It is difficult to modify and one tends to avoid changing it very much. In fixing bugs, it would have been ten-fold better to have writable microcode.

Debugging was enhanced with the use of a breakpoint box that could breakpoint in the microcode. The APU code could also be breakpointed. A system DDT allowed us to monitor various tables as well as look at the data structures at will.

At one point there was major difficulty when queues were being clobbered. Since there were only forward pointers, it was hard to reconstruct the situation. The use of backward pointers as well would certainly enhance the integrity of the queue structures. (It was later found that the queues were being clobbered by a hardware bug). Checks to ensure the integrity of the queue could thus be inserted.

The statistics provided are very few. But what little there are prove very useful. By varying system load, one can tell how often the AMC skips swaps because there is not enough time to set up (startup). A larger emphasis on statistics would provide much improved measurement capability.

The present implementation of the memory manager is not capable of supporting 500 users. But it has gone a long way towards that goal. Most important, it confirms that that goal is attainable. The use of a powerful

memory management processor (and probably a swapping processor) in a multiprocessor organization holds great promise in achieving a large-scale utility supporting 500 users.

Appendix I ABBREVIATIONS FOR MODEL I MEMORY MANAGEMENT SYSTEM

AMC	auxiliary memory controller
AMTU	auxiliary memory transfer unit
BLK	process status bit: process blocked
CB	context block
CBC	process status bit: context block read is queued
CHT	core hash table
CWS	core working set
DEST	this DHT entry is being destroyed
DHT	drum hash table
DIRTY	dirty bit in CHT
DKT	drum to disk transfer
DKW	this DHT entry is being written on disk
DSQ	drum sector queue
DWIP	drum write in progress bit in CHT
DWS	drum working set
EC	error count in AMC node
ECD	error code for disk transfers, in PRT
EOS	end of sector
ERR	error field in PMT
FCL	free core list

FP	file page
FR	functional registers in TSU
GREQ	general request queue
HNE	header not equal flag
HR	holding registers in TSU
IB	index block
K	disk address
KCQ	disk cylinder queue
KDT	disk to drum transfer
KSQ	disk sector queue
KVALID	disk copy of this DHT entry is valid
KW	disk write bit in CHT
LDD	process status bit: Process loaded
MIB	multiple index block
MMS	memory management system
MSQ	process status bit: on μscheduler queues
OKL	OK to load TSU registers
OP	operation code for TSU instruction
PC	position counter in TSU
PM	physical map of a CPU
PMT	process memory table
PP	private page
PQ	process status bit: page reads are queued
PRT	process table

PWS	position within sector
R	rotation time for drum or disk in ms
RIP	read in progress bit in CHT
RN	real name
RUN	process status bit: process running
SC	scheduled count in CHT
SF	scheduled flag in PMT
SWAPQ	swapping request queue
SWQ	process status bit: read request is on SWAPQ
TSU	transfer sub-unit
TUIM	transfer unit interface multiplexer
UC	use count
UN	unique name
UNAVL	unavailable bit in CHT
UNID	object identification field of unique name
UNTAG	tag field of unique name
UNUSER	user field of unique name
WL	write list
WUN	this DHT entry needs to have its UN written on disk

Appendix II MICROPROCESSOR GENERAL THEORY OF OPERATION

A. Introduction

The BCC Microprocessor is a synchronous, 24-bit digital computer. The flow of data between the functional elements of the computer is controlled by terms generated by the microprocessor 90-bit instruction word. Table A1, below, is a listing of the bits in the instruction word and a definition of their function.

1. Data Flow. (See Fig. 1)

The X-bus and the Y-bus are the two principal intra-processor data transfer busses. Data transfer between the microprocessor and ancillary devices is accomplished by the E1-bus and the E2-bus for a parallel-input (pin), and by the Z-bus for a parallel-output transfer (pot). Data transfer between the microprocessor and the core memory is accomplished by the M1-bus and the M2-bus for input data, and by the M-bus for output data. The data busses are 24-parallel transfer lines gated by terms derived from the microprocessor instruction word.

2. Read-Only Memory (ROM). (See Fig. 2)

The ROM is a diode memory containing the 90-bit instruction words. The ROM is addressed by the O register and outputs the selected 90-bit instruction word to the I register.

3. I Register.

The I register is a 90-bit register and contains the current microprocessor instruction being executed. The I register is normally loaded at the end of each machine cycle, from the ROM.

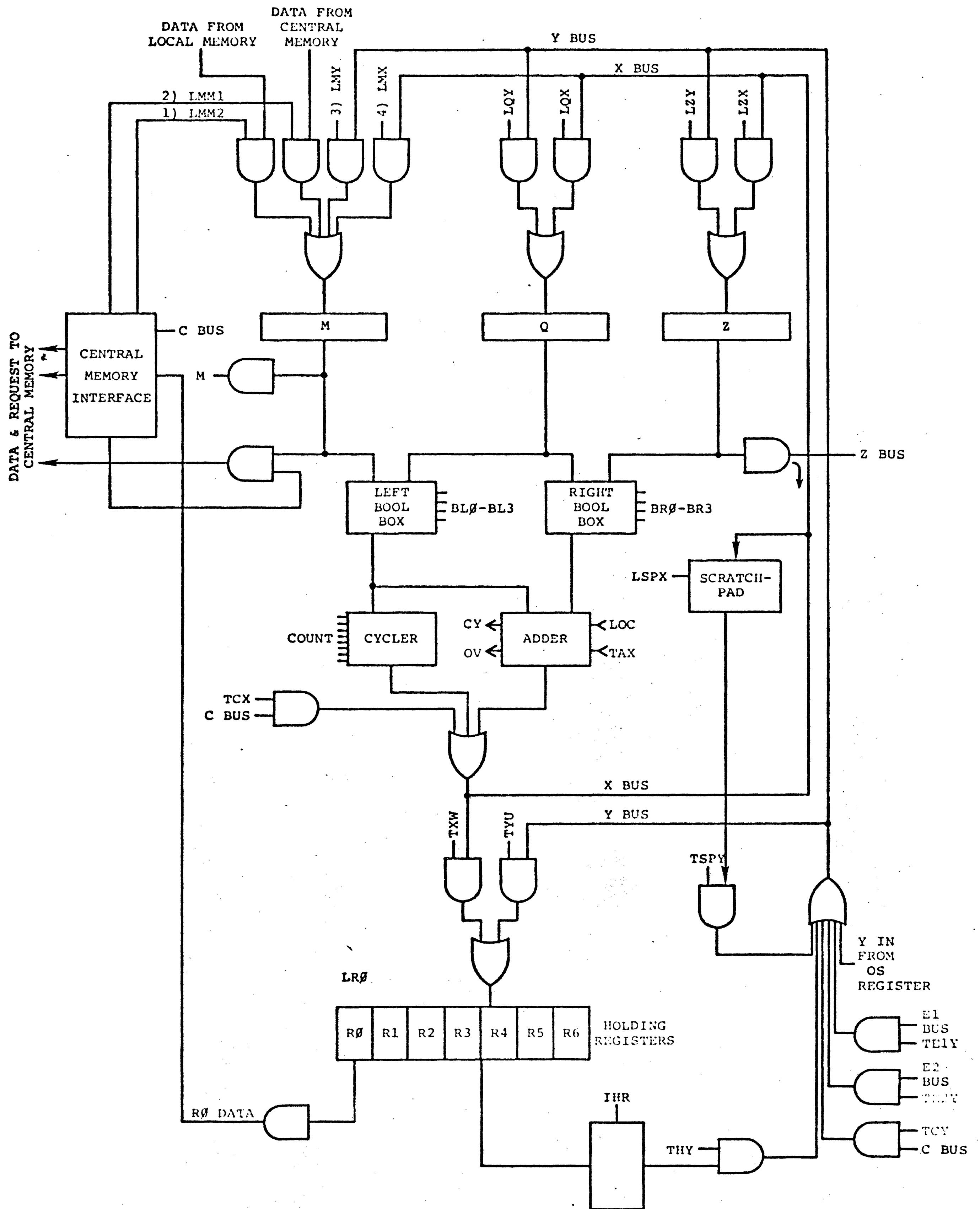
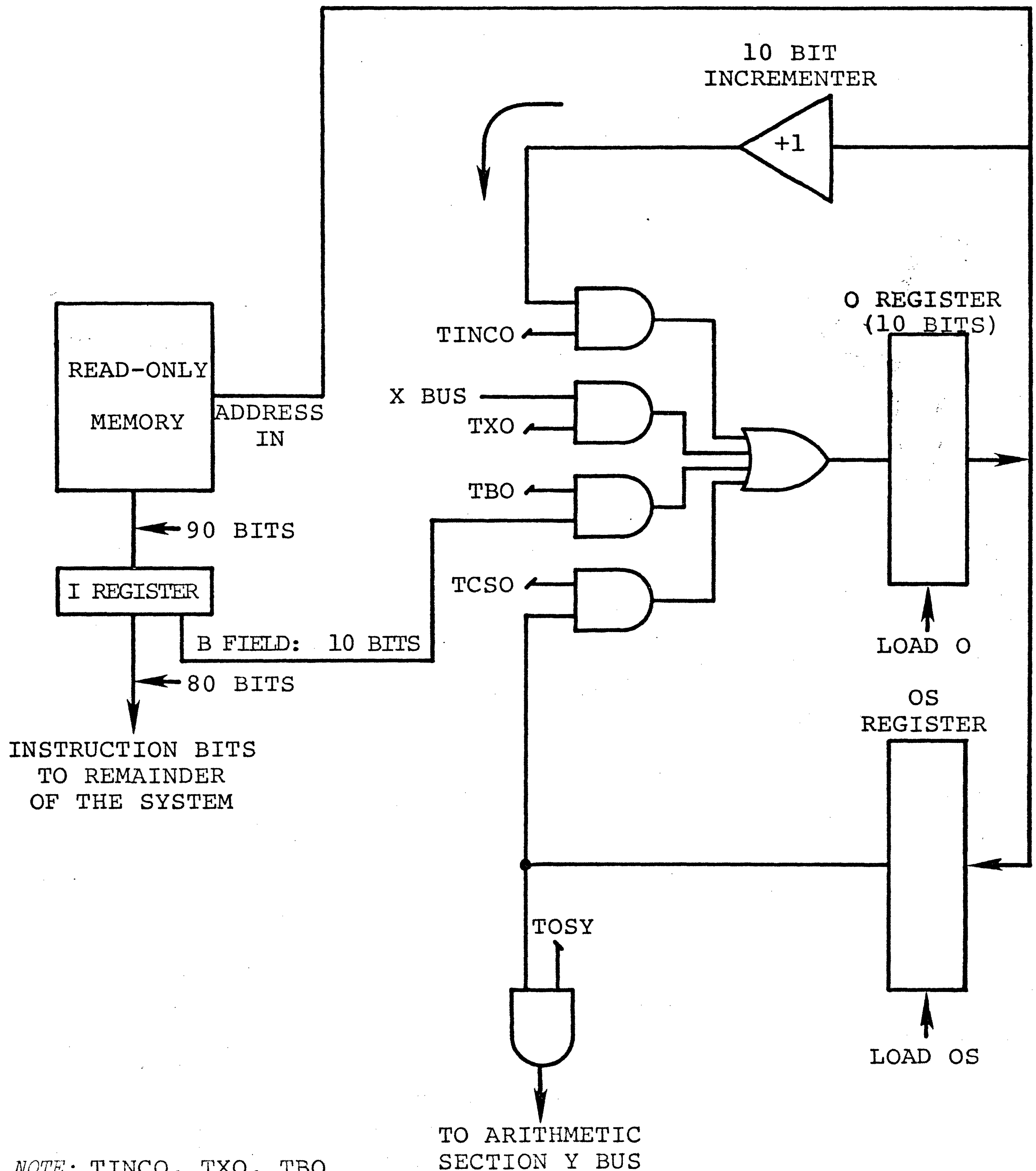


Figure 1; Microprocessor Data Paths: Arithmetic Section



NOTE: TINCO, TXO, TBO and TCSO are generated by the control circuitry.

Figure 2: MICROPROCESSOR DATA PATHS; CONTROL SECTION

4. <u>O Register</u>.

The O register is a 10-bit register which holds the address of the instruction word to be executed in the next cycle. It can be loaded from the B-field of the instruction word, the least significant 10 bits of the X-bus; the incremented contents of the O register; or the OS register.

5. <u>OS Register</u>.

The OS register is a 10-bit register used to save the return address for subroutine calls. The contents of the OS register can be transferred to the Y-bus.

6. <u>M, Q, and Z Registers</u>.

The M, Q, and Z registers are 24-bit registers, and are loaded from the X-bus or the Y-bus. The M register is also loaded, independently, from the local or central memory under control of the central memory interface. The buffered outputs of the Z register are used for parallel-output-transfers (pot) to ancilliary devices via buffers on the I/O interface card. Two boolean (bool) boxes associated with the M, Q, and Z registers provide inputs to the adder/cycler. The output of the left bool box is any one of the sixteen logical functions of M and Q; the output of the right bool box is any one of the sixteen logical functions of Z and Q. The logical functions to be performed are specified by the BR and BL fields of instruction word. See Table A4 for details.

7. Holding Registers.

The holding registers, RØ thru R6, are 24-bit registers which are loaded from the X-bus or the Y-bus. The output of the holding register can be incremented, and is gated by the instruction word to the Y-bus.

8. Scratchpad.

The scratchpad is a 24-bit by 64-word IC memory, loaded from the X-bus, and read into the Y-bus and addressed either by a field in the instruction word or by the least significant six bits of the Z register.

9. Adder/Cycler.

The adder portion of the adder/cycler is a 24-bit full-adder with an anticipated carry. The adder sums the output of the left and right bool boxes. The resultant sum is transferred to the X-bus. A low-order-carry input to the adder may be generated directly by a bit in the instruction word. The cycler portion of the adder/cycler is controlled by the instruction word or by the Z register, and left cycles the output of the left bool box to the X-bus.

10. Typical Instruction Cycle.

Every instruction in the microprocessor is a conditional branch. The MCONT field (2 bits) of the instruction word specifies the location of the branch address as either the Branch Address field of the instruction, the X-bus, or the OS register. The MC field of the instruction word specifies one of 64 conditions which, if satisfied, will cause a branch to occur. If the branch condition is not satisfied, the contents of the O register (present address incremented) are used as the address of the

the next instruction word; and at the end of the machine cycle, the next instruction is fetched from the ROM and the O register is incremented. When the branch condition is satisfied, the O register is not incremented at the end of the machine cycle; the register clocks are inhibited. The cycle time of a successful conditional branch is, therefore, extended in the sense that the normal overlap of instruction execution with instruction fetching is abrogated. A success causes the O register to be loaded from the source specified by the MCONT field of the instruction word. During the extended interval, the O register fetches the word in the branch destination address.

B. Microprocessor Instruction

A microprocessor instruction may require one, two, or three machine cycles to be executed. The control logic contains two flip-flops, XXB and XXC, which comprise the state counter. The state counter determines from VCY, DGO, and BRANCH in the instruction whether an instruction will take one, two, or three machine cycles. The three possible states are state A (XXB'·XXC'), state B (XXB·XXC'), and state C (XXB' XXC). The length of time required to complete an instruction depends on the type of instruction.

1. Unsuccessful Branch Instructions.

These instructions do not branch and do not have the VCY bit in the instruction set. Execution of the instruction occurs in state A and requires only one machine cycle.

2. Stretched Unsuccessful Branch Instructions.

These instructions do not branch and have the VCY bit in the

instruction set. Execution of the instruction occurs at the end of state B and, therefore, requires two machine cycles. State A is a waiting period that allows signals to propogate through long paths such as scratchpad, adder, and tests of X.

3. Successful Branches.

These are instructions where the branch condition is satisfied. They require two machine cycles and, therefore, use both state A and state B. Register loading is done at the end of state B, but the O register is loaded with the branch destination address at the end of state A. At the end of state B, the O register is loaded again, this time with the branch destination address plus one. Simultaneously, the I register is loaded with the instruction contained at the branch address.

4. Stretched Successful Branch Instructions.

These are instructions for which the branch condition is satisfied, and the VCY bit in the instruction is set. These instructions use three machine cycles and, therefore, require states A, B, and C. This type of instruction is used when the branch address or condition requires the time to be generated. Loading of any register specified in the instruction occurs at the end of state C.

5. Subroutine Calls.

A subroutine call stores the contents of the O register (the address of the instruction being executed plus 1) in the OS register, and loads the O register with the subroutine address. This instruction requires two machine cycles, state A and state B, and must, therefore, have the VCY bit set in the instruction.

6. Deferred Branch Instruction.

Deferred branch instructions cause the instruction after the deferred branch instruction (current address plus 1) to be executed before the branch occurs. In order to execute a deferred branch, the DGO bit in the instruction is set. This instruction uses state A only and requires one machine cycle. If the VCY is set in the instruction, an additional machine cycle will be available to prepare the branch condition or address, and the instruction will use state A and state B. In a deferred branch, the O register is loaded from the B-field of the instruction, the least significant ten bits of the X-bus, or the OS register at the end of state A if VCY is set.

Table A1. 90-bit Microinstruction Word

Signal	Position	Clock	Function
MCØ-MC5	0-5	K2	Branch Condition field (6 bits). (See Table A2).
MCONTØ,1	6,7	K3	Branch control field (2 bits): Ø = branch conditionally to the address specified by the contents of BØ thru B9. 1 = branch conditionally to the address specified by the contents of BØ thru B9. Store the contents of the 0 Register (return address) in the OS Register. 2 = branch conditionally to the address specified by the contents of the OS Register. 3 = branch conditionally to the address specified by the contents of the X-bus ten lsb.
BØ, B1, B2 B3, B4, B5 B6, B7, B8 B9	8,9,10 11,12,13 14,15,16 17	K3	Branch address field (10-bits).

Table A1. 90-bit Microinstruction Word, (cont'd)

Signal	Position	Clock	Function
CØ, C1, C2	18,19,20	K3	Constant Field (24-bits).
C3, C4, C5	21,22,23		
C6, C7, C8	24,25,26		
C9, C10,	27,28,29		
C11, C12,	30,31,32		
C13, C14,	33,34,35		
C15, C16,	36,37,38		
C17, C18,	39,40,41		
C19, C20,			
C21, C22,			
C23			
IHR	42	K2	Increment holding register.
TCX	43	K3	Transfer the contents of C-field to the X-bus.
TCY	44	K3	Transfer the contents of C-field to Y-bus.
TSPY	45	K3	Transfer the contents of the selected scratchpad address to the Y-bus.
THY	46	K3	Transfer the contents of the holding register to the Y-bus.

Table A1. 90-bit Microinstruction Word. (cont'd)

Signal	Position	Clock	Function
TXW	47	K3	Transfer the data in the X-bus to the holding register.
TYW	48	K3	Transfer the data in the Y-bus to the holding register.
TAX	49	K3	Transfer adder output to the X-bus.
LOC	50	K2	Low order carry to adder.
SSPØ-5	51-56	K2	Select the contents of scratchpad address $Ø-77_{(8)}$.
TOSY	57	K3	Transfer the contents of the OS register to Y-bus (bits 14-23).
LRØ	58	K3	Load holding register RØ.
LSPX	59	K3	Load the selected scratchpad address from the X-bus.
MSØ-MS5	60-65	K2	Special functions field. (See Table A3.)
RRNØ-2	66-68	K2	Holding register select read field (enables selected holding register, RØ thru R6, output).
LRNØ-2	69-71	K3	Holding register select load field (clocks selected holding register, R1 thru R6, input).

Table A1. 90-bit Microinstruction Word. (cont'd)

Signal	Position	Clock	Function
LMX	72	K3	Loads the M register from the X-bus.
LMY	73	K3	Loads the M register from the Y-bus.
LQX	74	K3	Loads the Q register from the X-bus.
LQY	75	K3	Loads the Q register from the Y-bus.
LZX	76	K3	Loads the Z register from the X-bus.
LZY	77	K3	Loads the Z register from the Y-bus.
BLØ-BL3	78-81	K2	Left bool box control field (See Table A4).
BRØ-BR3	82-85	K2	Right bool box control field (See Table A4).
VCY'	86	K3	State counter set-term.
DGO	87	K3	State counter set-term.
TE1Y	88	K3	Transfer data from the E-1 Bus to the Y-bus.
TE2Y	89	K3	Transfer data from the E-2 bus to the Y-bus.

Table A2. Branch Conditions

MCØ-MC5	Branch Conditions
ØØ	Never branch
Ø1	Always branch
Ø2	X=Ø
Ø3	X≠Ø
Ø4	X<Ø
Ø5	X≥Ø
Ø6	X>Ø
Ø7	Y≥Ø
1Ø	Y<Ø
11	RØ<Ø
12	RØ≥Ø
13	X≤Ø
14	X'∧777777B=Ø, (X(6)-X(23)=777777B)
15	X'∧777777B≠Ø, (X(6)-X(23)≠777777B)
16	Z≥Ø
17	Z<Ø
2Ø	Always Branch
21	Y∧7≠Ø, (Y(23)∨Y(22)∨Y(21)=1
22	BL=Ø
23	BL≠Ø
24	Y(23)=Ø
25	Y(23)≠Ø

Table A2. Branch Conditions (cont'd)

MCØ-MC5	Branch Conditions
26	Attention latch 1=Ø ⚠1
27	Request Strobe latch 1 = Ø and Request Strobe latch 2 = Ø
3Ø	Protect≠X
31	Request Strobe latch 2 = Ø
32	Special flag A=Ø
33	Special flag A≠Ø
34	Attention latch 2=Ø ⚠1
35	Attention latch 3=Ø ⚠1
36	Attention latch 1≠Ø ⚠1
37	Not decoded
4Ø	Undefined
41	Undefined
42	Local memory parity error=1 ⚠1
43	Undefined
44	Central memory parity error=1 ⚠1
45	Breakpoint≠Ø
46-77	⚠2

⚠1 Resets latch.

⚠2 46 thru 77 not decoded.

Special Functions

Each microprocessor has several special functions, principally concerned with I/O. These functions are controlled by the MSØ-MS4 field of the microinstruction. Some have branch conditions associated with them which may be tested with the MCØ-MC4 field of the instruction.

POT/PIN

The POT/PIN System allows the microprocessor to communicate with external devices. When a microprocessor wishes to transfer data to an external register, it puts the address of the external register to be loaded (hardware defined) into the Z Register and sends an Alert strobe to all external devices. The external device takes the address from the Z bus, and uses it to set up a path from the Z bus to the specified register. The microprocessor will then load Z with the data to be sent, and send a 'POT' strobe. The external device will use this POT signal to load the selected register from Z. When an external device wishes to send data to a microprocessor, it sends an 'ATTENTION' signal to the microprocessor. This signal is latched in the microprocessor and may be tested by a branch condition. The microprocessor can then read a register in the external device by sending an 'ALERT' to the device. The device will set up a path between the selected register and the E bus. The microprocessor will then transfer the E bus to the Y bus, use the data, and send a 'PIN' strobe signifying that it has read the data.

Request Strobe

Each microprocessor has one latch which can be set by the other microprocessors in the system. This latch may be tested by a branch condition.

A unit can also selectively set the latches in all the other micro-processors by gating the contents of the X bus to the request strobe lines. X will contain an 8 bit mask which will determine which of the other microprocessors are to be strobed. It is legal for a microprocessor to strobe itself.

Table A3. Special Functions

MSØ-MS5	Function
ØØ	No activity
Ø1	LCY1
Ø2	LCY2
Ø3	LCY3
Ø4	LCY4
Ø5	LCY8
Ø6	LCY12
Ø7	LCY16
1Ø	LCY2Ø
11	LCL Z (CCFZA)
12	LCH Z (CCFZB)
13	SKZ (SPFZ)
14	ALERT
15	POT
16	PIN
17	Request Strobe #1
2Ø	Unprotect
21	Unusable
22	LPF
23	Reset Request Strobe Latch #1 [1]
24	Reset Central Memory Request [2]
25	Request Protect
26	Reset T.U.
3Ø	Set special flag A
31	Reset special flat A
32	Reset Request Strobe Latch #2
33	Request Strobe #2
34	Undefined
4Ø	Release [3]

Table A3. Special Functions (cont'd)

MSØ-MS5	Function
41	Prestore △3
42	Store △3
43	Store & Hold △3
44	Fetch △3
45	Fetch & Hold △3
47	Prefetch △3
6Ø	Set Bank B
61	Set Bank A
62	Clear all CPU Maps
64	Fetch △4
65	Fetch & Hold △4
	△5

△1 Occurs at end of instruction

△2 Local and Central Memory

△3 Memory Reference

△4 ODDWORD FETCH

△5 Special functions 27, 35, 36, 37, 46, 63, and 66 thru 77 are not decoded

Bool Box Functions

The bool boxes generate the 16 possible functions of 2 variables in response to their control fields. BLØ-BL3 controls the left bool box (functions of M and Q), BRØ-BR3 controls the right bool box (functions of Z and Q). The functions are:

BLØ-BL3 or BRØ-BR3	Left Bool Box Output	Right Bool Box Output
0	$M \cdot Q$	$Z \cdot Q$
1	$M = Q$	$Z = Q$
2	Q	Q
3	$\overline{M}+Q$	$\overline{Z}+Q$
4	M	Z
5	$M+\overline{Q}$	$Z+\overline{Q}$
6	$M+Q$	$Z+Q$
7	1	1
10	Ø	Ø
11	$\overline{M} \cdot \overline{Q}$	$\overline{Z} \cdot \overline{Q}$
12	$\overline{M} \cdot Q$	$\overline{Z} \cdot Q$
13	$\overline{M}$	$\overline{Z}$
14	$M \cdot \overline{Q}$	$Z \cdot \overline{Q}$
15	$\overline{Q}$	$\overline{Q}$
16	M(EOR)Q	Z(EOR)Q
17	$\overline{M}+\overline{Q}$	$\overline{Z}+\overline{Q}$

Table A4. Bool Box Control

BLØ-BL3	Left Bool Box Output
ØØ	$M \cdot Q$
Ø1	$M=Q$
Ø2	Q
Ø3	$\overline{M}+Q$
Ø4	M
Ø5	$M+\overline{Q}$
Ø6	$M+Q$
Ø7	1
1Ø	Ø
11	$\overline{M} \cdot \overline{Q}$
12	$\overline{M} \cdot Q$
13	$\overline{M}$
14	$M \cdot \overline{Q}$
15	$\overline{Q}$
16	M(EOR)Q
17	$\overline{M}+\overline{Q}$

BRØ-BR3	Right Bool Box Output
ØØ	$Z \cdot Q$
Ø1	$Z=Q$
Ø2	Q
Ø3	$\overline{Z}+Q$
Ø4	Z
Ø5	$Z+\overline{Q}$
Ø6	$Z+Q$
Ø7	1
1Ø	Ø
11	$\overline{Z} \cdot \overline{Q}$
12	$\overline{Z} \cdot Q$
13	$\overline{Z}$
14	$Z \cdot \overline{Q}$
15	$\overline{Q}$
16	Z(EOR)Q
17	$\overline{Z}+\overline{Q}$

Appendix III AMC Startup

There is a convenient facility for starting up the AMC. Instead of loading from paper tape or magnetic tape, the AMC can be initialized with information kept on reserved areas on the drum and disk. This startup procedure is controlled by the use of a Switch Register. It is one word which is located at 5 in absolute memory. It contains fields which give the following information (see Fig. A5):

a. AMC or CHIO initialization.

b. Store memory onto crash area.

c. TSU number.

d. Choice of 2 devices attached to TSU.

e. Two bits which are interpreted as described below to give the band address on the device.

There are two algorithms for computing the starting address for the crash area. This one address then determines both the crash and system areas. One of the algorithms pertains to the drum and one pertains to the disk. The critical thing is that the two devices have a different number of sectors on one band. It takes two bands on the drum to store the system (32 sectors) and two more to keep the virgin system. On the disk it will take 6 bands for the crash area and 6 for the system all on cylinder (track) zero:

BIBLIOGRAPHY

1. Denning, P.J., "The Working Set Model for Program Behavior," Comm. of the ACM, Vol. 11, No. 5, 1968.

2. "BCC Microprocessor Manual", BCC Corporation, 1970.

3. Freeman, Jack, "Process Memory System," Manual, Internal Documentation, BCC Corporation, 1970.

4. Lampson, B.W., "Memory Management System," Specifications, Internal Documentation, BCC Corporation, 1970.

5. "Specifications for a High-Performance Auxiliary Memory System," BCC Corporation, 1969.

6. Van Tuyl, R.R., "AMC Phase 1 Notes," Working Papers, Internal Documentation, BCC Corporation, 1970.

7. Van Tuyl, R.R., "An Algorithm for Swapping Data from Drum to Core," Master's Thesis, University of California, Berkeley.