SOFTWARE MANUAL

AMOS MONITOR

CALLS MANUAL

DWM-00100-42

AMOS
monitor calls
manual

This document reflects AMOS versions 4.1 and later

'AMOS', 'AlphaBASIC', and 'AM-100'

are trademarks of products
and software of

ALPHA MICROSYSTEMS
Irvine, CA 92714

ALPHA MICROSYSTEMS
17881 Sky Park North
Irvine, CA 92714

PREFACE

One of the major features of the AMOS operating system is the large number of monitor calls available to the assembly language programmer. By making most common routines available in the monitor, AMOS frees the programmer from having to repetitivly write the same routine. This manual is designed to describe these monitor calls.

It is assumed that the reader of this manual is familiar with assembly language programming and the AM-100 instruction set. It is also assumed that the reader is familiar with the AM-100 macro assembly system described in the "Assembly Language Programmers Manual." This manual is most emphatically NOT a tutorial on assembly language programming. Many such tutorials exist; if the reader is just learning assembly language, he or she should consult such a book before reading this manual.

Table of Contents

CHAPTER 1 COMMUNICATING WITH THE AM-100 MONITOR

1.1 MONITOR CALL CALLING FORMAT 1-1
1.1.1 Arguments 1-2
1.1.2 Standard Address Arguments 1-2

CHAPTER 2 JOB SCHEDULING AND CONTROL SYSTEM

2.1 THE JOB CONTROL BLOCK (JCB) 2-1
2.1.1 Example - Scanning The Job Control Area 2-2
2.2 ACCESSING YOUR JCB 2-3
2.2.1 Calling Sequence 2-3
2.3 JOB SCHEDULING CALLS 2-3
2.4 JOB CONTROL BLOCK FORMAT 2-3
2.4.1 JOBSTS - The Job Status Word 2-4
2.4.2 JOBSPR - The Stack Pointer Reset Address 2-4
2.4.3 JOBNAM - The Job Name 2-4
2.4.4 JOBBAS - The Memory Base Address 2-5
2.4.5 JOBSIZ - The Memory Partition Size 2-5
2.4.6 JOBUSR - The Current PPN 2-5
2.4.7 JOBPRV - The Privilege Word 2-5
2.4.8 JOBPRG - The Current Program Name 2-5
2.4.9 JOBCMZ - The Command File Size 2-6
2.4.10 JOBCMS - The Command File Status 2-6
2.4.11 JOBERC - The Error Control Address ... 2-6
2.4.12 JOBTYP - The Job Type 2-6
2.4.13 JOBBPT - The Breakpoint Address 2-7
2.4.14 JOBBNK - The Memory Bank Pointer 2-7
2.4.15 JOBDEV - The Default Device 2-7
2.4.16 JOBDRV - The Default Drive 2-7
2.4.17 JOBTRM - The Terminal Block Pointer .. 2-7
2.4.18 JOBRBK - The Run Control Block 2-8
2.4.19 JOBFPE - The Floating-Point Trap Address 2-8
2.4.20 JOBRNQ - The Scheduling Area 2-8
2.4.21 JOBDYS - The DYSTAT Address 2-9
2.4.22 JOBSTK - The Job's Stack Area 2-9

CHAPTER 3 MEMORY CONTROL SYSTEM CALLS

3.1 MEMORY PARTITION FORMAT 3-2
3.2 MEMORY MODULE FORMAT 3-5
3.3 MANIPULATING MEMORY MODULES 3-6
3.3.1 Allocating a Memory Module 3-8

3.3.2 Changing a Memory Module 3-8
3.3.3 Deleting a Memory Module 3-8
3.3.4 Permanent and Temporary Modules 3-8
3.4 MEMORY MAPPING SYSTEM 3-9
3.4.1 Internal Table Format 3-10
3.4.1.1 The MEMDEF Word 3-10
3.4.1.2 The JOBBNK Word 3-11
3.4.2 The Bank Switching Process 3-11
3.4.3 The BNKSWP Monitor Call 3-11

CHAPTER 4 LOADING AND LOCATING MEMORY MODULES

4.1 THE SRCH AND FETCH CALLS 4-1
4.1.1 Specifying the Module Name 4-1
4.1.2 The Module Address 4-2
4.1.3 Flags 4-2
4.1.3.1 F.FCH - Fetch Module From Disk 4-2
4.1.3.2 F.USR - Bypass System Memory Search 4-3
4.1.3.3 F.ABS - Bypass Memory Search 4-3
4.1.3.4 F.FIL - Mark Module as Permanent 4-3
4.1.4 Completion Codes 4-3

CHAPTER 5 MONITOR QUEUE SYSTEM CALLS

5.1 INCREASING THE AVAILABLE QUEUE LIST SIZE 5-1
5.2 QUEUE BLOCK USAGE BY THE SYSTEM 5-2
5.3 QUEUE SYSTEM MONITOR CALLS 5-3
5.3.1 QGET - Obtain a Free Queue Block 5-3
5.3.2 QRET - Return a Queue Block 5-3
5.3.3 QADD, QINS - Manipulating Queue Blocks 5-3

CHAPTER 6 THE FILE SERVICE SYSTEM

6.1 THE DATASET DRIVER BLOCK 6-1
6.1.1 DDB Format 6-2
6.1.1.1 Error Code 6-2
6.1.1.2 Flags 6-4
6.1.1.3 Buffer Address 6-4
6.1.1.4 Record Size 6-4
6.1.1.5 Buffer Index 6-4
6.1.1.6 Record Number 6-5
6.1.1.7 Queue Chain Link 6-5
6.1.1.8 JCB Address 6-5
6.1.1.9 Job Priority 6-5
6.1.1.10 Device Code 6-5
6.1.1.11 Drive 6-5

6.1.1.12 Call Level 6-6
6.1.1.13 Filename and Extension 6-6
6.1.1.14 PPN 6-6
6.1.1.15 Open Code 6-6
6.1.1.16 Driver Work Area 6-6
6.1.2 Device Transfer Buffers 6-6
6.1.3 Error Handling 6-7
6.1.3.1 Error Codes 6-7
6.2 FILE SERVICE MONITOR CALLS 6-8
6.2.1 FSPEC - Process an ASCII Filespec 6-8
6.2.2 INIT - Initialize the DDB 6-9
6.2.3 LOOKUP - Find the File 6-10
6.2.4 OPENI - Open a File for Input 6-10
6.2.5 OPENO - Open a File for Output 6-11
6.2.6 OPENR - Open a File for Random Processing 6-11
6.2.7 CLOSE - Close a File 6-11
6.2.8 READ - Perform a Physical Transfer 6-11
6.2.8.1 Sequential Devices 6-11
6.2.8.2 Random Devices 6-12
6.2.8.3 Interrupt Structure 6-12
6.2.9 WRITE - Perform a Physical Write 6-12
6.2.9.1 Sequential Devices 6-12
6.2.9.2 Random Devices 6-12
6.2.9.3 Interrupt Structure 6-13
6.2.10 INPUT - Perform a Logical Read 6-13
6.2.10.1 Sequential File Processing 6-13
6.2.10.1.1 Example 6-14
6.2.10.2 Random File Processing 6-14
6.2.10.3 Special Devices 6-14
6.2.11 OUTPUT - Perform a Logical Write 6-14
6.2.11.1 Sequential File Processing .. 6-15
6.2.11.1.1 Example 6-15
6.2.11.2 Random File Processing 6-15
6.2.11.3 Special Devices 6-15
6.2.12 WAIT - Wait for IO Completion 6-15
6.2.13 DELETE - Delete a File 6-16
6.2.14 RENAME - Rename a File 6-16
6.2.15 ASSIGN - Assign a Device 6-16
6.2.16 DEASGN - Deassign a Device 6-17
6.3 DISK SERVICE MONITOR CALLS 6-17
6.3.1 Calling Sequence 6-18
6.3.2 The Bitmap Area 6-18
6.3.2.1 The Status Word 6-18
6.3.2.2 The Bitmap DDB 6-18
6.3.2.3 The Bitmap Buffer 6-19
6.3.2.4 The Bitmap 6-19
6.3.2.5 Altering the Bitmap 6-19
6.3.3 DSKCTG - Allocate a Contiguous Area ... 6-19
6.3.4 DSKALC - Allocate a Record 6-20

6.3.5 DSKDEA - Deallocate a Record 6-20
6.3.6 DSKBMR - Read the Bitmap 6-20
6.3.7 DSKBMW - Write the Bitmap 6-20
6.3.8 DSKDRL - Lock the Directory 6-21
6.3.9 DSKDRU - Unlock the Directory 6-21

CHAPTER 7 TERMINAL SERVICE SYSTEM

7.1 TERMINOLOGY 7-1
7.2 THE TERMINAL LINE TABLE 7-2
7.2.1 The Terminal Status Word 7-2
7.3 THE TERMINAL SERVICE CALLS 7-2
7.3.1 KBD - Fetch a Line of Data 7-2
7.3.2 TTY - Output One Character 7-3
7.3.3 TIN - Get an Input Character 7-3
7.3.4 TOUT - Output One Character 7-3
7.3.5 TAB - Output One Tab 7-4
7.3.6 CRLF - Output a Carriage-Return / Line-Feed 7-4
7.3.7 TTYI - Output a String of Characters 7-4
7.3.8 TTYL - Output a String of Characters Indexed 7-4
7.3.9 PTYIN - Place Character in Input Buffer 7-5
7.3.10 PTYOUT - Fetch Character from Output Buffer 7-5
7.3.11 TTYIN - Fetch Another Jobs Input 7-5
7.3.12 TTYOUT - Place a Character in Another Jobs Output 7-5
7.3.13 TRMICP - Process Input Character Within Interface Driver 7-5
7.3.14 TRMOCP - Process Output Character Within Interface Driver 7-6
7.3.15 TRMBFQ - Process Output Characters Within Terminal Driver 7-6
7.3.16 TBUF - Output Large Amounts of Data ... 7-6
7.3.17 TCRT - Call Special Terminal Driver Routines 7-7
7.3.17.1 Standard Functions 7-7
7.3.17.1.1 Cursor Addressing 7-7
7.3.17.1.2 Other Functions 7-7
7.3.18 Message Calls 7-8

CHAPTER 8 CONVERSION MONITOR CALLS

8.1 NUMERIC CONVERSION CALLS 8-1
8.1.1 Calling Format 8-1
8.1.1.1 Size Byte 8-1

8.1.1.2 Flags 8-2
8.2 RAD50 CONVERSION MONITOR CALLS 8-3
8.2.1 RAD50 Packing Algorithm 8-3
8.2.2 Packing and Unpacking Calls 8-3
8.2.2.1 PACK - Pack Three ASCII Characters into RAD50 8-4
8.2.2.2 UNPACK - Unpack Three RAD50 Characters into ASCII 8-4
8.3 PRINTING CONVERSION CALLS 8-4
8.3.1 PFILE - Output a Filespec From a DDB .. 8-4
8.3.2 PRNAM - Output a Filename 8-4
8.3.3 PRPPN - Output a PPN 8-5

CHAPTER 9 INPUT LINE PROCESSING CALLS

9.1 ALF - TEST A CHARACTER FOR ALPHABETIC 9-1
9.2 NUM - TEST A CHARACTER FOR NUMERIC 9-2
9.3 TRM - TEST A CHARACTER FOR TERMINATOR 9-2
9.4 LIN - TEST A CHARACTER FOR LINE TERMINATOR ... 9-2
9.5 BYP - BYPASS BLANKS 9-2
9.6 GTDEC - INPUT A DECIMAL NUMBER 9-2
9.7 GTOCT - INPUT AN OCTAL NUMBER 9-2
9.8 GTPPN - INPUT A PROJECT-PROGRAMMER NUMBER 9-3
9.9 FILNAM - INPUT A FILENAME 9-3

CHAPTER 10 MISCELLANEOUS MONITOR CALLS

10.1 EXIT - RETURN TO AMOS COMMAND LEVEL 10-1
10.2 SLEEP - PUT JOB TO SLEEP 10-1
10.3 CTRLC - BRANCH ON CONTROL-C 10-2

APPENDIX A DISK STRUCTURE FORMAT

A.1 PHYSICAL RECORD FORMAT A-1
A.2 DISK RECORD TYPES A-2
A.2.1 The Disk ID Record A-2
A.2.2 The Bitmap A-2
A.2.3 The Master File Directory A-3
A.2.4 The User File Directory A-3
A.2.5 Sequential File Data Records A-3
A.2.6 Contiguous File Data Records A-3
A.3 FILE STRUCTURE A-3
A.4 MFD ITEM FORMAT A-5
A.5 UFD ITEM FORMAT A-5

APPENDIX B SYSTEM COMMUNICATION AREA

B.1 SYSTEM - SYSTEM ATTRIBUTES WORD B-1
B.2 DEVTBL - ADDRESS OF THE DEVICE TABLE B-1
B.3 DDBCHN - ACTIVE DDB CHAIN B-2

B.4 MEMBAS & MEMEND - USER MEMORY POINTERS B-2
B.5 SYSBAS - BASE OF SYSTEM MEMORY B-2
B.6 JOBTBL - ADDRESS OF THE JOB TABLE B-2
B.7 JOBCUR - JCB ADDRESS OF THE CURRENT JOB B-2
B.8 JOBESZ - JOB TABLE ENTRY SIZE B-3
B.9 TIME - THE TIME OF DAY B-3
B.10 DATE - THE SYSTEM DATE B-3
B.11 HLDTIM - THE HEAD LOAD TIMER B-3
B.12 CLKFRQ - LINE CLOCK FREQUENCY B-4
B.13 SPXSAV - STACK POINTER SAVE LOCATION B-4
B.14 SPXINT - INTERNAL STACK B-4
B.15 LPTQUE - LINE PRINTER SPOOLER QUEUE B-4
B.16 TRMDFC - BASE OF THE TERMINAL DEFINITION TABLE B-4
B.17 TRMIDC - ADDRESS OF FIRST INTERFACE DRIVER .. B-4
B.18 TRMTDC - ADDRESS OF FIRST TERMINAL DRIVER ... B-5
B.19 TRMSCN - THE NON-INTERRUPT TERMINAL QUEUE ... B-5
B.20 CLKQUE - THE CLOCK QUEUE B-5
B.21 SCNQUE - THE IDLE SCAN QUEUE B-5
B.22 RUNQUE - THE JOB SCHEDULING QUEUE B-5
B.23 DRVTRK - THE DRIVE/TRACK TABLE B-5
B.24 MEMDEF & MEMBNK - MEMORY MANAGEMENT CONTROL . B-6
B.25 ZSYDSK - ADDRESS OF SYSTEM DISK DRIVER B-6
B.26 QFREE - QUEUE SYSTEM CONTROL B-6

APPENDIX C ALPHABETICAL LISTING OF AMOS MONITOR CALLS

INDEX

CHAPTER 1

COMMUNICATING WITH THE AM-100 MONITOR

The AM-100 monitor contains over 70 routines which are available for use by assembly language programs running in user or monitor memory. These routines are called by the supervisor calls SVCA and SVCB which have been coded into macro form to make them easy to incorporate into user programs. The macros are included as a part of the system library file SYS.MAC in area [7,7] of the system disk. These calls have been grouped into logical sections according to the function they perform and will be described in this and the following chapters.

1.1 MONITOR CALL CALLING FORMAT

The general format for all monitor calls is:

{label:} opcode {arguments} {;comments}

As can be seen from the format, the only required item in all calls is the opcode itself which is the name of the monitor call. A label may be used if desired in which case it is assigned the address of the SVCA or SVCB instructions which start all monitor call sequences. The total number of words generated by any monitor call depends upon the call itself. Some calls generate up to four words of code to perform the function. Those calls which incorporate an ASCII message (such as the TYPE call) will generate a string of bytes which will vary in length depending on the message involved. Comments may also be placed at the end of the line as desired just as in the machine instructions and will be preceded by the semicolon which is the identifying character for all comments.

1.1.1 Arguments

Some calls will require one or more arguments which specify parameters for the execution of the monitor call function. These arguments most normally are source and/or destination address items for the data being manipulated by the monitor call. Some calls allow the user to specify the location of data parameters while other calls operate with predefined registers that must be set up by the user beforehand. As each call is defined in the sections that follow the arguments that are required will be detailed. The arguments will normally be defined as being expression values, standard addresses, or ASCII strings. Expression values may be any valid source expression which evaluates down to a value which is within the predefined range of the argument definition. ASCII strings are just that; a string of characters which are probably used as a message to be displayed. Standard addresses are so important and complex that we will devote the next whole section to the explanation of them.

1.1.2 Standard Address Arguments

NOTE

> The following section is one of the most important, and most frequently misunderstood, sections of this manual. The concept of standard arguments is fundamental to understanding the monitor call calling sequences.

Standard addresses form the heart of many of the more complex monitor calls and therefore should be thoroughly understood by the user in order to gain maximum flexibility from the system. A standard address argument is coded exactly the same as a standard source or destination operand for a machine instruction such as ADD or MOV. There are some restrictions that should be noted, however, due to the method used in processing the standard address. Standard addresses are only used with those monitor calls that are coded as SVCB instructions. The SVCB pushes all user registers onto the stack and it is from these stored values on the stack that the monitor call processor gains access to the address calculations using those registers. Standard addresses may take the form of any of the legal WD16 addressing modes however all autoincrement and autodecrement processing is done on a word basis even though the monitor call may be requesting only one byte of data. In addition, the value used for SP register references is a dummy value which is not reloaded into SP when the monitor call exits so the autoincrementing and autodecrementing modes will be ignored if used with the stack pointer register.

The monitor call processing software within the monitor actually duplicates the hardware and calculates the target address from the stored register value on the stack and the data from the extra word if the address mode uses

one. This target address then becomes the address of the data to be manipulated by the specific monitor call routine itself. Sometimes this data is only one byte while other times it may be several words or more. The target address calculated by the processing of the standard address argument always points to the first byte of the data if more than one byte is required by the monitor call. A special case occurs when the standard address argument specifies the direct register address mode. In the WD16 hardware instructions there is never more than one full word of data involved for the standard source and destination address modes, so direct register will work on either the low byte or the full word in the target register. In the processing of monitor call standard addresses, however, this is not always the case since we pointed out that some calls require several words of data to be manipulated. When direct register mode is used, the target address is actually the address of the stored register on the stack, which was a direct result of the SVCB hardware instruction processing. If more than one word is used by the call it will merely sequence right on through the stored words on the stack. In simple terms this means that if a monitor call wants three words of data for an argument and you specify the register R2 as the standard address argument, the three words that will be used will actually be the words in R2, R3 and R4, in sequence. This is often very useful when writing reentrant code. A word of caution: if you specify a register for a call that wants more words than you have registers, (most I/O calls want a 20-word DDB argument) the monitor call will walk right on through your stack and most likely crash the entire system.

One of the more common errors is forgetting that a standard argument needs a pound-sign (#) in front of a literal argument. For example, if we wish to sleep for 20 clock ticks, the code reads:

```
SLEEP   #20.
```

Note that without the pound-sign, we would sleep for the number of ticks contained in program-relative location 20.

At this point you should go back and reread the above section. It is very important that you understand the concepts outline above. Just try to think of the standard address arguments as good old source or destination addresses as in the machine instructions. When you foul them up you will definitely find out about it quickly, since the usual result is a system crash.

CHAPTER 2

JOB SCHEDULING AND CONTROL SYSTEM

The AMOS timesharing monitor performs the task of allocating jobs and scheduling CPU time and resources for their operation. In order to properly write assembly language programs which make use of some of the more complex features of the system, a basic understanding of how jobs are scheduled and controlled is necessary. The total theory of how jobs are handled is too great a task to attempt to cover in one section of this manual but it is hoped that enough information can be given to provide the basic fundamentals on job control by user programs.

Each job that is running in the system basically has two dedicated components which are not shared by any other job in the system; a monitor job control block and a user memory partition. In the monitor memory area itself there exists a job control table which contains one area for each job that has been allocated to the system. One job is allocated for each JOB command in the system initialization command file which gives the job name and the terminal to which it is connected. The area that is allocated for each job in the job control table is used to contain specific information about that job. This area is called the job control block and will be referred to from now on as the JCB.

2.1 THE JOB CONTROL BLOCK (JCB)

The format of the JCB is defined in the system library file SYS.MAC as a series of equate statements. Each equate statement has the name JOBxxx where xxx is a three-character code for the specific item of the JCB being defined. The value of this symbol is actually the offset in bytes from the base of the JCB to the item itself. The user may, during the course of his program, desire to read the current data in his own JCB or in some instances modify it. References to the JCB items should be made in one of two ways:

1. Use the system monitor calls JOBGET, JOBSET and JOBIDX which is the preferred method.

2. Locate the JCB for your job by moving @#JOBCUR into a register and then referencing all JCB items via JOBxxx(Rx).

There are three words in the system communication area which define the entire job control system during timesharing operation. These three words are not part of the JCB areas but rather are non-sharable parameters set up during system initialization and not part of any one job. I point this fact out because the names of these three words are JOBTBL, JOBCUR and JOBESZ which appear to be part of a user JCB but really are not. JOBTBL contains the base of the JCB table where all JCB's are stacked sequentially. This address is set up at system initialization time and is never changed. JOBCUR always contains the address of the JCB which has current control of the CPU and is updated to point to the new JCB each time the job scheduler switches to a different job. Therefore, @#JOBCUR will always point to your JCB if you reference it because the reference will only be executed while you have current control of the CPU. JOBESZ contains the size of the JCB in bytes and is used by the system and by user programs for scanning through the JCB table. Since the size of the JCB may expand as new features are added to the system, JCB table scans must be made by setting an index to the base of the table (MOV @#JOBCUR,Rx) and then adding the size to the index to get to the next entry (ADD @#JOBESZ,Rx). When scanning the JCB table, the first word of each JCB is guaranteed to be non-zero and the table is terminated by a null (zero) word. Again, these three words are a part of the master system communication area and not in the job table itself.

2.1.1 Example - Scanning The Job Control Area

The following is a brief example of how to scan the JCB table and process each JCB entry (such as for a system status report):

```
        MOV     @#JOBTBL,R0     ;set JCB table index R0 to table base
;Loop here to process each job table entry (JCB)
LOOP:   ...     ...             ;process JCB entry which is indexed by R0
        ...     ...             ;references to JCB items are via JOBxxx(R0)
        ...     ...
        ADD     @#JOBESZ,R0     ;advance R0 to next JCB entry
        TST     @R0             ;is this end of JCB table?  (null word)
        BNE     LOOP            ;  nope - go process valid JCB entry
;At this point we have finished the job table scan
        ...     ...
        ...     ...
```

2.2 ACCESSING YOUR JCB

There are three monitor calls which should be used to gain access to your own JCB when necessary. Two calls are used to transfer a single word of data to and from a specific word in the JCB and one call is used to set an index to a specific spot in the JCB area so that multiple words may be transferred or so that faster access may be obtained when needed.

```
JOBGET  tag,item        ;Transfers one word from JCB item to tag
JOBSET  tag,item        ;Transfers one word from tag to JCB item
JOBIDX  tag,item        ;Sets absolute address of JCB item into tag
```

2.2.1 Calling Sequence

All calls share the same basic format where tag is a standard argument used for the transfer of one word of data in the JOBGET and JOBSET calls or to receive the index address in the JOBIDX call. The item argument is one of the JCB item tags (JOBSTS, JOBPRI, JOBNAM, etc.) which identifies the desired item to be used in the transfer or to have the index set to. These items are equated to their relative offset value in SYS.MAC and will be explained in the remainder of this section along with the usage of each and its importance to the user, if any.

2.3 JOB SCHEDULING CALLS

In addition to the above calls, there are two calls used by various routines within the system monitor for controlling the job scheduling processes. These calls are JWAIT and JRUN which will set a job into the wait state and then reactivate it to the run state, respectively. Both calls require that the job being controlled be indexed by R0 (which must point to the base of the JCB for that job) and that the argument specify one of the status control bits (in JOBSTS) to be used as the control flag. The user should not make use of these calls so no more detail will be given here.

2.4 JOB CONTROL BLOCK FORMAT

The following is a list of the entries contained in your JCB. Each of these entries may be accessed via JOBGET, JOBSET, or JOBIDX by using the tag defined in each entry.

2.4.1 JOBSTS - The Job Status Word

The first word in each JCB is the job status flag word. Each bit in this word indicates a particular state in which the job may reside. Some legal states are defined by more than one bit being on at a time. The system and some of the system programs set and reset these bits as the current state of the job changes but the user is cautioned against altering this word unless extreme caution (and intelligence) is used. A brief list of the bits and the mneumonics assigned to them follows along with a basic description of the function of the bit when it is set.

```
J.ALC=1         ;Job entry is allocated (guarantees JOBSTS non-zero)
J.TIW=2         ;Job is in Terminal Input Wait state
J.TOW=4         ;Job is in Terminal Output Wait state
J.SLP=10        ;Job is in Sleep state
J.IOW=20        ;Job is in I/O Wait state
J.EXW=40        ;Job is in External Event Wait state
J.CCC=200       ;A control-C abort is waiting to be processed
J.RUN=400       ;Job is running
J.MON=1000      ;Job is in monitor command mode (no program active)
J.LOD=4000      ;Program is being loaded for execution
J.SUS=10000     ;Job is in Suspend state
J.LOK=20000     ;Job has CPU locked (by user program command)
```

If any of the following flags are on the job will not be scheduled for CPU run time until the flag has been cleared: J.TIW, J.TOW, J.SLP, J.IOW, J.EXW, or J.SUS.

2.4.2 JOBSPR - The Stack Pointer Reset Address

One word which is used to store the stack pointer reset address which is calculated when the system is initialized. This address is then used to reset the stack pointer each time the job exits back to monitor command mode. The user may allocate a larger stack area within his own partition by reloading this address if desired.

2.4.3 JOBNAM - The Job Name

Two words which contain the six-character job name packed RAD50. This name is set up by the JOBS command in the system initialization file. If a user program alters this word it is effectively altering the name of the job.

2.4.4 JOBBAS - The Memory Base Address

One word which contains the base address of the user memory partition if one has been allocated for this job. This address is altered only by the MEMORY program which allocates and deallocates user memory partitions. The user is advised against altering this address unless a thorough understanding of the memory allocation process is first attained.

2.4.5 JOBSIZ - The Memory Partition Size

One word which contains the size of the user memory partition in bytes if one has been allocated for this job. This size word together with the above JOBBAS address word define the current user memory partition. JOBSIZ is altered only by the MEMORY program and the monitor command processor.

2.4.6 JOBUSR - The Current PPN

One word which contains the current user PPN (account number) if the user is logged in. Zero indicates that no user is currently logged into this job. JOBUSR is modified by the LOG and LOGOFF programs and is tested by various protection schemes in the system to allow user access to files, etc.

2.4.7 JOBPRV - The Privilege Word

One word which is used to store the privileges associated with the job. This word is not currently used but is allocated for future implementations of the security system. Further documentation will be provided when the system is completed.

2.4.8 JOBPRG - The Current Program Name

Two words which contain the six-character program name which is currently running or was the last job run if in monitor command mode. JOBPRG is loaded with the program name (packed RAD50) by the command processor when the program is loaded or located for execution. Currently the only significance of this program name is in the displays created by the SYSTAT program (user terminal status display) and the DYSTAT program (video monitor).

2.4.9 JOBCMZ - The Command File Size

One word which contains the size of the current command file area in the user memory partition if a command file is being processed. If this word is zero no command file is currently in effect. This word is set to the initial size of a command file when that file is loaded into the top of the user partition and is decreased as each line is extracted from the area and sent to the monitor command processor. When it gets to zero the command file is finished and the system returns to normal command mode input from the user terminal. The user should not alter this word.

2.4.10 JOBCMS - The Command File Status

One word which contains flags used by the command file processor when a command file is being processed. These flags should never be altered by the user so they will not be detailed here. JOBCMS works in conjunction with JOBCMZ to effect the command file processing scheme.

2.4.11 JOBERC - The Error Control Address

One word which controls the processing of WD16 hardware bus errors as described in the "WD16 Programmer's Reference Manual." If JOBERC is zero a bus error will cause a message to be printed on the user terminal and the job will be aborted. If JOBERC is non-zero a jump will be made to the address specified in JOBERC which had better contain a valid routine for gracefully shutting down the program. Note that the bus error is fatal for this user only and does not normally kill the whole timesharing system.

2.4.12 JOBTYP - The Job Type

One word which specifies the type of job which is assigned to this jobstream. The following flags are currently implemented:

```
J.USR=1         ;Job is a user partition
J.NUL=2         ;Job is currently running the null subroutine
J.NEW=4         ;Job is processing a new memory allocation
J.LPT=10        ;Job is running the line-printer spooler (LPTSPL)
J.HEX=20        ;Binary inputs and outputs are in hex (not octal)
J.DER=40        ;Print disk error retry messages
J.VER=100       ;Activate auto-verify mode for disk writes
J.CCA=200       ;Control-c interrupts are enabled
J.GRD=400       ;Terminal is guarded against SEND commands
```

2.4.13 JOBBPT - The Breakpoint Address

One word which specifies the address to jump to if a breakpoint is encountered during the execution of a user program. JOBBPT is used by the DDT debug program for breakpoint handling and not normally used by user programs.

2.4.14 JOBBNK - The Memory Bank Pointer

One word which is used by the memory management system to define the bank in which the job's current memory partition resides. It is actually a pointer to the control item within the memory mapping table which is used for turning the bank on and off when the job is allocated CPU time. This word must not be modified by the user.

2.4.15 JOBDEV - The Default Device

One word which contains the RAD50 device code for the default device to be used if the file specification being processed by the FSPEC call does not explicitly specify a device. Normally this default device is DSK.

2.4.16 JOBDRV - The Default Drive

One word which contains the drive number in binary for the default drive number to be used if the file specification being processed by the FSPEC call does not explicitly specify a drive number. Only used if the device code matches the code in JOBDEV or if the device code is left to default also. JOBDEV and JOBDRV normally contain the device and drive number set by the LOG program when a user logs in. They specify the disk device and drive which will usually be used for processing done by this user.

2.4.17 JOBTRM - The Terminal Block Pointer

One word which contains a pointer to the terminal definition block for the terminal which is currently attached to this job. If no terminal is currently attached, this word contains a zero. The first word in the terminal definition block is the terminal status word which is available to the user for modification to set various terminal parameters such as echo control, image mode and lower case processing. The old monitor call TIDX would deliver the address of this status word back to the user in register R0. The TIDX call is no longer supported and must be replaced by the more general call:

```
        JOBGET  R0,JOBTRM       ;Get status word index
```

As with all of the JOBxxx calls, the destination may be any valid address and not just R0 as in the example above. The above example will replace the TIDX call exactly in performance since TIDX used R0 as its destination.

For further information on the format of the terminal definition block and its use the user should refer to the source listing of the terminal service routine (TRMSER) which is made available to users on a special source diskette, as well as on the standard system disk pack. The terminal definition block is defined at the beginning of this routine.

2.4.18 JOBRBK - The Run Control Block

A 14-word area which is the run control block for the jobstream. It is used for the loading of programs and overlays during job execution and is set up by the user program with the parameters needed to fetch the next program or overlay segment prior to the execution of a FETCH call. Refer to the description of the FETCH monitor call for more details on the use of this item.

2.4.19 JOBFPE - The Floating-Point Trap Address

One word which contains the address to jump to if a floating point error is executed such as a divide by zero. Used by AlphaBasic to control such nonsense. A user program which executes floating point instructions should enter his error trap address into JOBFPE and not into the vector at memory location 76 since this would destroy the sharable resource of that vector.

2.4.20 JOBRNQ - The Scheduling Area

A 7-word area which maintains the parameters for job scheduling and context switching of this job. The first four words are dynamically changing links which are used during the job scheduling process to place the job into the active run queue for future processing. The user must never alter any of these four link words without first taking out group health insurance.

The fifth and sixth words are used to determine the job's run priority. The fifth word (at JOBRNQ+10) is the time counter which is decremented once for each clock interrupt whenever the job is running. When this count goes to zero the job is put into the wait state and another job is activated. The sixth word (at JOBRNQ+12) is the actual priority of the job (set up by the JOBPRI command) and is used to initialize the above time counter each time the job is given control of the CPU for running. These two words replace the old system word called JOBPRI in the JCB.

The seventh word is used for storage of the current stack pointer value when the job is not in the active run state. The scheduler restores the stack pointer from this word each time the job is reactivated.

2.4.21 JOBDYS - The DYSTAT Address

One word which contains the address to the byte in the VDM screen memory area for the job execution arrow. Set by the DYSTAT program and referenced by the monitor job scheduler. The user should not alter this address.

2.4.22 JOBSTK - The Job's Stack Area

A 100-word area which acts as the stack for this job. SP is set to the top of this area when a new program is initiated. The user may reset his own stack pointer by moving the address of a larger area within his own partition if the program needs more stack area. Be sure to allow at least an extra 20 words or so for possible real-time interrupt handling which needs a valid stack area for register saves. The job scheduler also saves all user registers and some other nonsense on the user stack during job context switching.

The label "JOBSTK" is not defined explicitly in SYS.MAC but the area exists as the last 100 words in the JCB. The area has not been labeled because the JCB may be increased in size as the need arises and the JOBSTK area should not be referenced by a label which will change value in future releases.

CHAPTER 3

MEMORY CONTROL SYSTEM CALLS

The AM-100 system contains a fairly sophisticated memory control system even though there is no memory protection or mapping hardware associated with it. In order to make maximum use of the memory resources available and minimize system crashes due to memory violations the assembly language programmer should understand how the monitor allocates memory and the rules under which memory should be accessed. This section will describe the memory allocation scheme and the monitor calls that assist the user in using memory in the proper way.

The memory that is available to the AM-100 processor is up to 64K bytes (32K words) with the top 256 bytes being unavailable because it is mapped to the I/O ports. The AMOS monitor resides in low memory beginning at location zero and extending upward as far as the monitor requires (typically around 14K bytes). The remaining memory above the monitor up to the end of the total amount of memory in your system is available for assignment as user memory partitions for each of the jobs. All of the user memory may be allocated to one job or it may be split up into several partitions of varying sizes with one partition allocated to each job. The amount of memory that a user program has to play with is therefore defined as the single contiguous memory partition which has been assigned to his job by the operator MEMORY command. This memory partition block is then allocated into smaller defined blocks called "modules" which are used by the system and the user to contain programs and data areas. Monitor calls exist which allow the user program to locate the absolute boundaries of his own memory partition and also to allocate, change, and delete memory segments in the form of defined modules. These modules have the capability of being named just like files (filename.extension) so they may be located by that name. Any program loaded for execution will be in the form of a module. During execution some programs create other modules for device buffers, data tables, etc.

3.1 MEMORY PARTITION FORMAT

The memory partition assigned to a job may be located anywhere in memory depending on the memory that was available when the job assigned it using the MEMORY operator command program. The user program may not count on any specific location for this partition. Within the partition, memory modules are allocated upward beginning at the base of the defined partition and building modules on top of each other as long as space permits. Modules may not be built that will extend past the top boundary of the user partition. As modules are deleted from memory all modules above them are automatically shifted downward to fill up the space that the deleted module left. Also, when any module is changed in size the modules above it are shifted in position accordingly. This method insures that all available memory is always at the top of your partition in one contiguous block. This method of grabbing the first hunk of free memory to load a program into is the main reason that all programs must be written in totally relocatable code.

Figure 3-1 shows a typical memory layout for three users operating in a 64K system. The free memory at the 56K boundary could be used by a fourth job or by one of the current jobs for expansion.

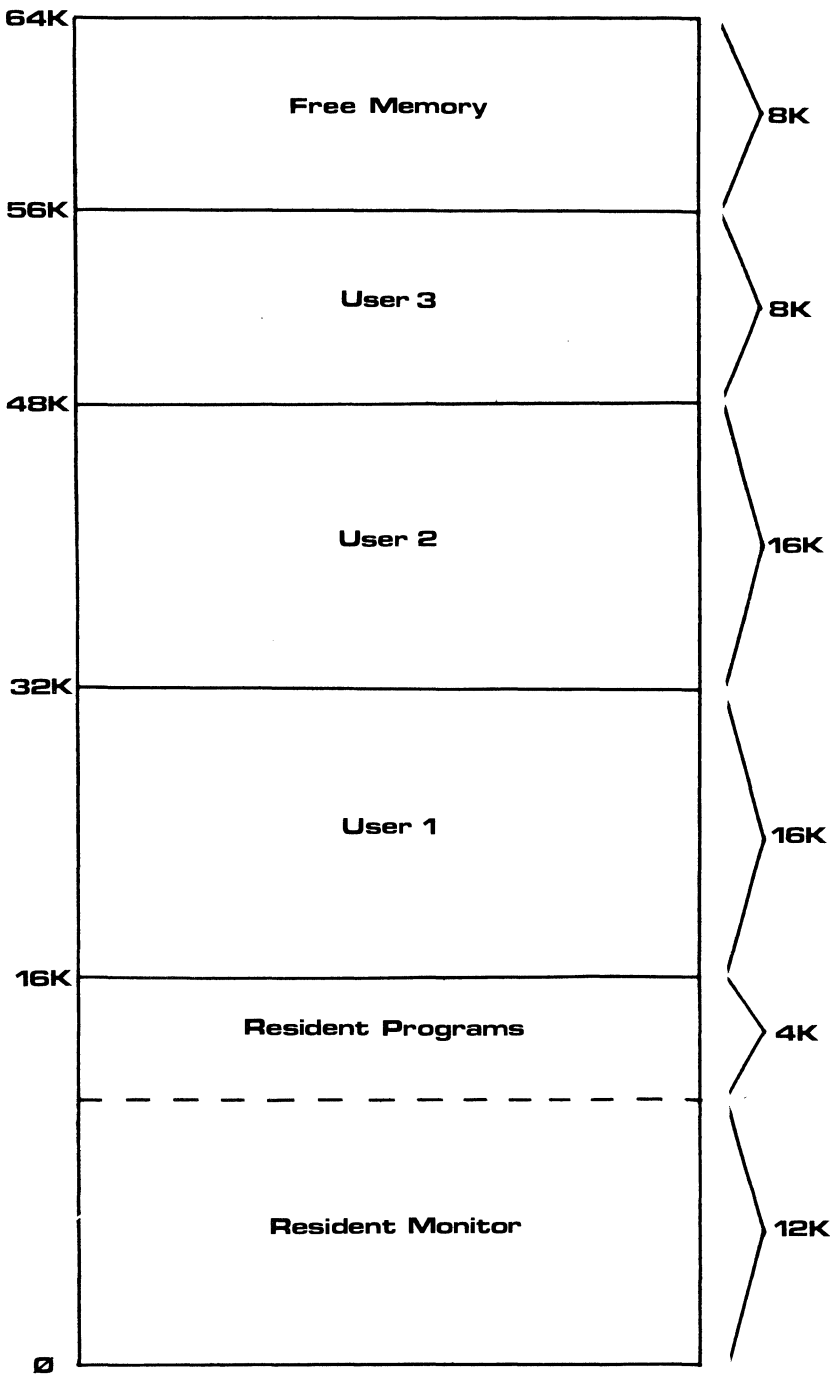
There are three monitor calls that will return information about your memory partition as it happens to be allocated. These three calls all take a single standard argument into which will be delivered the absolute address of the base, end, or free base of the user memory partition. The three calls and the addresses that they return are listed below:

```
USRBAS  arg  - absolute base of user memory partition (last word)
USREND  arg  - absolute end of user memory partition (last word)
USRFRE  arg  - current base of remaining free memory (last module+2)
```

Since modules must always occupy an even number of bytes the above calls will always return an even address. If no modules are allocated in the current partition the USRFRE address will equal the USRBAS address. Otherwise, the USRFRE address will be the word following the last currently allocated module in the memory partition. The remaining free memory that the user may make use of may be calculated by subtracting the USRFRE address from the USREND address.

Figure 3-2 shows a typical user job partition during the execution of a program which was loaded automatically by the operating system. The program itself was the first module to be allocated in the user partition and then was executed after being loaded. It will remain in memory until it has completed its task and exits to monitor at which time it will be deleted by the operating system monitor. During execution the program allocated a 1K data table module which possibly may be used for storage of symbols or some similar function. Two I/O files were then opened on disk which caused the operating system file service routine to allocate the two disk buffer modules. The remaining memory in the partition has not yet been allocated in our example.

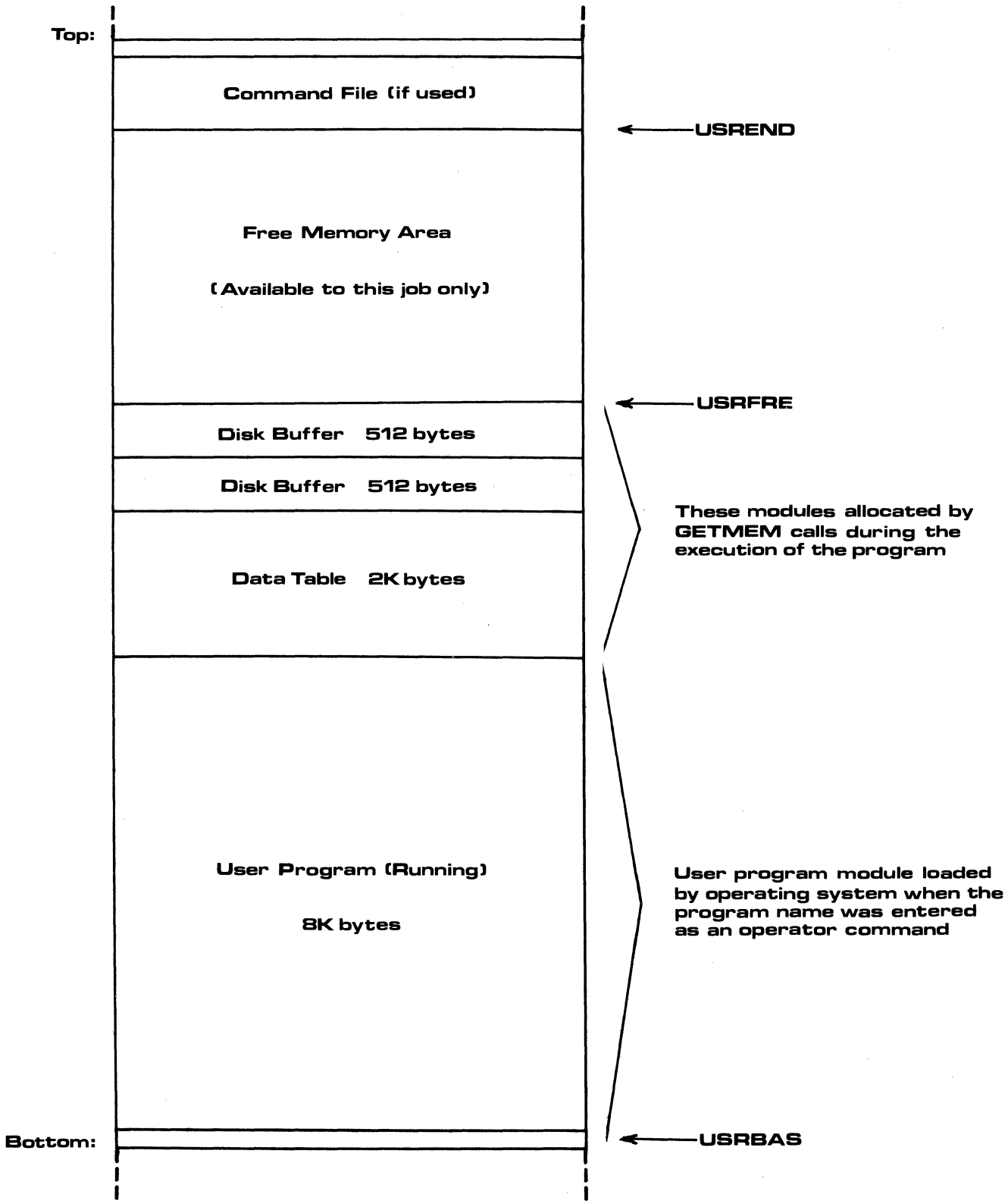
Note: Memory sizes are typical



Total resident monitor size is 16K, leaving 48K for user partitions

Memory Map for a Typical 64K System (3 users)

Fig 3-1



Memory Map for a Typical User Job Partition

Fig 3-2

Note that the USREND call does not actually return the absolute end of the partition but rather the end of the available free memory at the time of the call. If a command file is in progress it occupies the upper part of the partition which we do not wish to alter during the execution of a program. In fact, the program should not have to take into consideration whether or not it was called by direct command or from a command file. Use of the USREND call insures that the user program may use all of free memory without having to compensate for the remaining part of any command file module.

Although the standard use of memory by the operating system is through the use of the memory management system calls (to be described next) the user may find that it is easier to make use of free memory without regard to module boundaries, especially for use in variable length tables or hashing techniques. For this reason, the free memory space is always defined as the

area between the addresses returned by the USRFRE and USREND calls. Note that the initialization of files normally results in the allocation of a buffer module and the operating system allocates this buffer at the current setting of the USRFRE address and then updates that USRFRE address. Therefore, the user must be sure all I/O buffers and any work modules are allocated before freely using the memory above the USRFRE address. The INIT and FETCH calls both cause the indirect allocation of a memory module in addition to the direct allocation or alteration of modules by the GETMEM, CHGMEM and DELMEM calls.

3.2 MEMORY MODULE FORMAT

Memory modules are the basic unit of formal data structure within the user memory partition. They are always allocated on word boundaries and must contain an even number of bytes to maintain this format. The monitor calls will automatically pad an odd sized module with a null byte to even it up. All modules contain five housekeeping words followed by any number of data words from zero to the maximum size left in the user memory partition. The five housekeeping words are always allocated and so a single-word module really takes up six words of memory.

The module format is as follows:

Word 1 - total size of module in bytes including the housekeeping words
Word 2 - module flag word
Word 3 - module filename packed RAD50
Word 4 - module filename packed RAD50
Word 5 - module extension packed RAD50
Words 6 thru n - module data area

Figure 3-3 gives a pictorial view of the above standard module format. The data area is usually the only area with which the user is concerned and so all references are made from the base of this area. The SRCH and FETCH calls (to be described in a later section) will return this absolute address when locating or loading the requested module instead of the address of the

base of the housekeeping words. References to the housekeeping words should therefore be made via negative offsets relative to the data base address.

When scanning for a specific module or locating the end of the current module string, the user may set his index using the USRBAS call which will return the address of the size word of the first allocated module. He may then merely check the housekeeping words for the correct module name or other determining parameters and if the module is to be bypassed, add the size word to the index. This bumps the index to the next module allocated. The last module always has a zero word following it and the user must be careful not to destroy this zero word if he is manipulating free memory directly wihtout allocating it using the memory calls.

The module filename and extension follow the same format as the file names on disk if the module in memory is named. The name is optional and need be used only if the module is to be located by name at a later time.

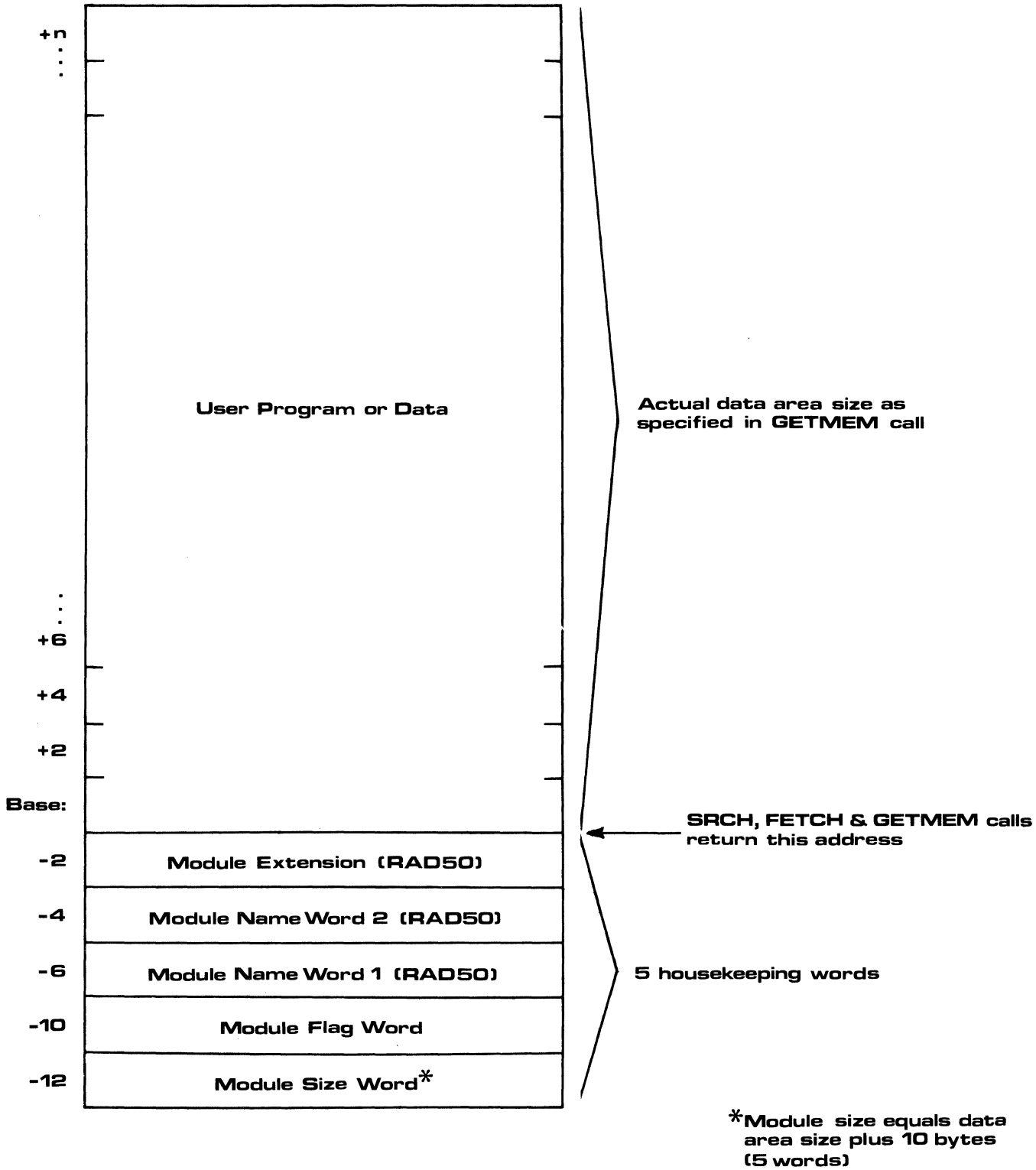
Modules may be either temporary or permanent depending on the method used to load them into memory. A module is made permanent by setting the file bit on in the housekeeping flag word when the module is allocated. Temporary modules are automatically deleted by the monitor when the program finishes and executes the EXIT call. Permanent modules are not automatically deleted but may be deleted by either the operator DELETE command or the monitor DELMEM call. Forcing a zero into the size word of the module is another way of deleting it but this is not the recommended way since this effectively also deletes all modules above it (the zero is the module area termination word).

3.3 MANIPULATING MEMORY MODULES

There are three monitor calls which are used to create, alter and delete these memory modules. All three calls take a single standard argument which must be the address of a two-word block called a memory control block (MCB). The first word of this MCB will contain the absolute memory address of the data area in the allocated module (past the housekeeping words). The second word will contain the size of the data area in bytes (ten bytes less than the total module size since the housekeeping words are not included). The MCB therefore is the user's block which defines a contiguous area in memory by its base address and size in bytes. The user need not be concerned with the housekeeping words unless he needs to access them directly for some exotic reason.

The following three calls are used to manipulate memory modules:

```
GETMEM  MCB   - allocates a new memory module at current USRFRE
CHGMEM  MCB   - changes the size of the module defined by MCB
DELMEM  MCB   - deletes the memory module defined by MCB
```



Standard Memory Module Format

Fig 3-3

3.3.1 Allocating a Memory Module

The following example shows the allocation of a 100 byte module

```
        MOV     #100.,MCB+2     ;set module size as 100 (decimal) bytes
        GETMEM  MCB             ;allocate module (MCB gets its address)
        ...
        ...
        ...
MCB:    WORD    0               ;receives address of module data area
        WORD    0               ;size of module data area in bytes
```

3.3.2 Changing a Memory Module

Then we may increase the size of the same module by

```
        ADD     #20.,MCB+2      ;increase size word by 20 bytes
        CHGMEM  MCB             ;change its size
```

The above will cause the monitor to adjust the module housekeeping size word to reflect the new size. The address of the module will not change but note that the USRFRE address will advance by 20 bytes and that any modules that were allocated after the one at MCB will be shifted up in memory but their corresponding addresses in their MCB will not be adjusted by the monitor. I/O buffers allocated after the MCB module will therefore be erroneously addressed after the change so the CHGMEM call must be used with care.

3.3.3 Deleting a Memory Module

To delete the above module we use the code

```
        DELMEM  MCB             ;delete the module
```

3.3.4 Permanent and Temporary Modules

Recall that all temporary modules will automatically be deleted by the monitor when the program exits. The user may force the module to be permanently left in memory by giving it a name and setting the file bit (defined in SYS.MAC as "FIL") in the flag word. The following example will illustrate the allocation of a 200 word module which is made permanent with the name "TABLE1.DAT":

```
        MOV     #200.,TBL1+2    ;set size as 200 bytes
        GETMEM  TBL1            ;allocate the module
        MOV     TBL1,R0         ;set R0 to index the data area base
        MOV     #[DAT],-(R0)    ;set the module name and extension (RAD50)
        MOV     #[LE1],-(R0)    ;  into the housekeeping words
        MOV     #[TAB],-(R0)    ;  in reverse order for efficient use of R0
        BIS     #FIL,-(R0)      ;set permanent file bit on in flag word
        ...
        ...
        ...
TBL1:   WORD    0               ;receives address of module
        WORD    0               ;size of module in bytes
```

Permanent memory modules may be saved onto disk using the operator SAVE command or they may be deleted from memory when done by the operator DEL command. Refer to the "AMOS User's Guide" for details on these commands.

3.4 MEMORY MAPPING SYSTEM

The AMOS system is capable of supporting memory in excess of 64K by a simple bank switching technique which turns selected memory boards on and off under control of the operating system. This section will attempt to define some of the technical aspects of that system. It is assumed that the reader has already familiarized himself with the operational aspects of the memory management system from the standpoint of setting up the SYSTEM.INI file commands and operating procedures.

The normal 64K memory area must be defined by the user for his own particular application as two general areas called sharable and switchable memory. Sharable memory always starts at location zero and extends upward far enough to totally contain the resident operating system and any system programs or sharable memory area needed for the application. Switchable memory then may occupy the remainder of the memory area up to the 64K address (octal 177376 inclusive).

There is only one sharable memory area which is always active. The switchable area, however, may be occupied by multiple memory boards referred to as "banks." Banks are defined to the operating system during system startup with the MEMDEF statements. Each MEMDEF statement defines the memory board (or boards) which are to be activated when that bank is selected by the operating system. Selection of the bank for activation is done when one of the user jobs which resides within that bank is granted CPU time by the AMOS job scheduling system. This action is automatic and transparent to the user. Only one bank may be active at any given point in time since all banks effectively respond to the same memory addresses (the area defined as switchable memory).

3.4.1 Internal Table Format

The memory bank switching system is controlled be a table which is dynamically built during system startup time by the MEMDEF statements. The table is basically a linked list of multi-word entries that resides within the monitor area. There is one entry which defines the sharable memory area and then there is one entry for each bank defined by a MEMDEF statement. Two words that reside in the monitor system communication area are used to control the memory management system. These words are labled "MEMDEF" and "MEMBNK" and are used to store the base address of the table just defined and the current memory bank which is active respectively. If memory management is not in use (no MEMDEF statements appeared in the SYSTEM.INI file) both of these words will contain a zero value.

3.4.1.1 The MEMDEF Word - The MEMDEF word in the system communication area will contain the address of the first entry in the table which will always be the entry defining the sharable memory boundaries. The format for this entry is:

Word 1 - link to next entry
Word 2 - base address of sharable memory (0)
Word 3 - top address of sharable memory plus 1

The remaining entries will define the switchable memory banks in use and have the format:

Word 1 - link to next entry (0 if this is last entry)
Word 2 - base address of this switchable bank
Word 3 - top address of this switchable bank plus 1
Words 4 through n - hardware control codes for bank switching

The hardware control codes are one or more entries which are used to turn the memory boards on and off during bank switching. There will be one control code for each physical board which has been defined as part of this bank. Each control code is two words in length with the first word containing the address of the hardware port for the memory board and the second word containing the switch-on and switch-off bytes (low and high bytes respectively) which are sent to that port. Note that in the MEMDEF statements you are able to specify more than one board per bank (even different types of boards) by separating the board definitions with slashes. The final hardware code will be followed by a single word of zero to indicate the end of the codes for this bank.

3.4.1.2 The JOBBNK Word - The JOBBNK word in each jobs JCB will contain the address of the word 4 in the above definition for the bank in which the job currently resides. This address is the base of the control codes for the hardware switching operation. The MEMBNK word in the system communication area will always contain the same address as the JOBBNK word for the job that is currently running. This is used by the scheduling and switching system to turn off the current job and turn on the next job for running.

3.4.2 The Bank Switching Process

Memory bank switching is performed by the job scheduler by a simple sequence of steps:

1. Use the MEMBNK word to locate the currently active bank entry.
2. Send the switch-off byte to the port address for each control code.
3. Use the JOBBNK word for the next job to be run to locate the bank entry for that job.
4. Send the switch-on byte to the port address for each control code.
5. Store the new job's JOBBNK data into the MEMBNK word for next time.

3.4.3 The BNKSWP Monitor Call

Under normal operation of the AMOS system each user is confined to an area that resides totally within any one defined memory bank. The BNKSWP call may be used by a more sophisticated assembly language routine to allow one user to access more than one bank of memory. The BNKSWP monitor call expects register R1 to contain the address of word 4 of the bank which is to be activated (similar to the automatic operation which uses the address within the JOBBNK word). The currently active memory bank will be switched off and the new bank (per R1 address) will be switched on. The MEMBNK word will be updated properly to reflect the newly activated memory bank. Register R1 will also be changed to contain the index to the previously operating bank thereby allowing a convenient return to reactivate the previous bank if R1 is not altered.

Note that since the current bank is switched off, the BNKSWP call must be executed from somewhere in sharable memory to prevent the return from executing instructions in the new bank (unless this is the exotic plan in mind). This can be accomplished in one of several different ways including pushing the routine onto your stack are (within the JCB) or executing a special subroutine which has been loaded into system memory.

CHAPTER 4

LOADING AND LOCATING MEMORY MODULES

Memory modules may contain an optional filename and extension, which may be used to locate modules, both in memory and on the disk. This chapter deals with locating and loading modules via these optional filenames and extensions. Normally, when an operator command is entered from the terminal, the first place searched for the requested program is the resident system memory area followed by the user's own memory partition. If the program is resident in either of these places, it need not be loaded in from disk, and execution begins immediately using the resident program directly where it was located.

4.1 THE SRCH AND FETCH CALLS

The user may make use of two monitor calls (FETCH and SRCH) for locating and loading modules in memory by name. In actuality, the SRCH call is a specialized version of the FETCH call and is included only for convenience and compatibility with older programs that are still hanging around in the system. Basically, the SRCH call will only locate a module if it is in memory while the FETCH call will automatically load a module into memory from the disk if it is not found to be in memory already.

Both calls have the same basic format:

```
SRCH    nameblock,index,control-flags
FETCH   nameblock,index,control-flags
```

4.1.1 Specifying the Module Name

Nameblock is a standard argument used in both cases to specify the name of the module to be located or loaded. The format of the actual nameblock referenced is different in each case, however. In the case of the SRCH call, nameblock refers to a 3-word block of memory (or 3 contiguous

registers) which contain the filename and extension of the desired module in RAD50 packed form. For the FETCH call, nameblock refers to a full file Dataset Driver Block (DDB) which allows the user to specify a full disk file specification to load the module from in case it is not located in memory. The DDB has not yet been introduced and will be defined and explained in a later section dealing with file I/O calls. In brief, the DDB is a 24 (octal) word area in memory which contains all the information and work areas to define and manipulate a specific disk file in any area on any defined disk device. The DDB is normally set up by processing an ASCII file specification with the FSPEC call (more on this later).

4.1.2 The Module Address

The second argument is the index which is to receive the absolute memory address of the located (or loaded) memory module data area. Refer to figure 3-3 in the preceding chapter for the layout of the memory module and the place that this index is set to. The index argument is also a standard argument although the normal mode is to receive the module address in a general register (R0-R5). If the index argument is not specified in the call, the default used is register R0 which is compatible with older versions of this system.

4.1.3 Flags

The third argument is the optional control flags which may be used to control the operation of the SRCH and FETCH calls. This argument is any valid expression which evaluates down to a value in the range of 0-17 (octal). Only the low order four bits are significant and they have been given the following mnemonic definitions in the system library SYS.MAC:

```
F.FCH=1         ;Fetch module from disk if not in memory
F.USR=2         ;Search user memory only
F.ABS=4         ;Load absolute segment from disk
F.FIL=10        ;Set module permanent file flag after load from disk
```

4.1.3.1 F.FCH - Fetch Module From Disk - F.FCH is the flag that actually differentiates the SRCH call from the fetch call since they both technically are the same SVCB supervisor call. The SRCH call forces this bit off while the FETCH call forces this bit on. When set, the F.FCH bit causes the nameblock to be interpreted as a full file DDB and the module to be loaded from disk if not located in memory first. Since the use of this bit is controlled by specifying either SRCH or FETCH as the calling opcode, the user is not supposed to include this bit in the control-flags argument of his call.

4.1.3.2 F.USR - Bypass System Memory Search - F.USR is the flag used to specify bypassing the searching of the resident system memory area for the module and proceed directly to searching the user area only. This allows specific versions of modules to be loaded and used even though they may possibly be duplicated in the system memory area. This flag is not normally used by programs other than system software.

4.1.3.3 F.ABS - Bypass Memory Search - F.ABS when set forces a direct search to the disk for the requested module, bypassing all memory searches that would normally occur. The module is then loaded into memory at the absolute address specified by the index argument in the calling sequence. No housekeeping words are allocated and the first word of the module gets loaded into the first word specified by the index argument. Note that this form is the only time that the index argument is used to pass an address to the FETCH processor instead of being used to receive the address of the located module. The F.ABS form of the FETCH call is used to load program segment overlays but I am certain that some other exotic uses will be thought of by you all out there.

4.1.3.4 F.FIL - Mark Module as Permanent - F.FIL is used to force the permanent file flag bit on in the module flag word after the module has been loaded from disk. The FETCH call always places the filename and extension into the housekeeping words 3-5 so even if the module is only temporary, it may still be located by name as long as the program which loaded it is still active. This is useful for dynamic loading of subprograms and/or data modules. Setting the F.FIL flag on in the control-flags argument means that the module will not be deleted from memory by the operating system when the calling program finally exits. The operator LOAD command uses this method to load a program into memory and leave it there to be called by name.

4.1.4 Completion Codes

When the SRCH or FETCH call returns, the user must test the status of the Z-bit to see if the module was located or loaded successfully. If the Z-bit is set (tested by BEQ) the operation was successful. If the Z-bit is not set (tested by BNE) the module was either not located or would not fit into the remaining free memory within the user's partition.

CHAPTER 5

MONITOR QUEUE SYSTEM CALLS

The monitor queue is a list of blocks in system memory which are linked to each other in a forward chain. The base of this chain, and the count of the blocks in the chain, are contained in the QFREE monitor communications words (See Appendix B). Each queue block in the chain links to the next one by storing its address in the first word of the queue block. The last queue block in the chain contains a zero link word to flag it as the end. Each queue block is currently 8 words (16 bytes) in size although this value may increase with the next release of the file system. The monitor initially contains 20 blocks in the available queue list.

During normal monitor operation various functions will use these queue blocks to perform certain tasks. When a routine needs a queue block it will issue a QGET monitor call which will deliver the first available queue block by returning its base address in register R3. The routine will then use this area to temporarily store information during processing. When the routine no longer requires the block it will issue a QRET monitor call which will return the queue block to the available list for later re-use.

The monitor queue system was necessary to provide storage for interrupt driven hardware (AM-300 board) and for storage during memory management operations. The queue blocks will always reside in sharable system memory and therefore may be used by interrupt routines without regard for memory management context switching. The monitor queue system will be used more and more as the monitor is improved but is also available to the user if desired. The XLOCK subroutine (for multi-user locks in AlphaBasic) uses the monitor queue system to store the lock parameters.

5.1 INCREASING THE AVAILABLE QUEUE LIST SIZE

It is apparent that the number of queue blocks in use at any one time will vary with system loading, number of users, and tasks being performed. Some applications may demand a larger available list of queue blocks to insure safe system operation. Due to overhaed restrictions, no check is performed

to see if the available queue is exausted and if this happens the system will probably begin doing strange and wonderful things. Therefore, there is a method by which you may increase the size of the available queue list during system startup time.

The monitor is initially generated with 20 free blocks in the available queue. At any time in the SYSTEM.INI file prior to the final SYSTEM command you may execute the QUEUE nnn command which will allocate "nnn" more queue blocks for general use. A typical increase for a large system with several users running extensive applications might be 100 more blocks for a total of 120.

Once the system is up and running no more queue blocks may be added to the list so you must give your best guess at your total requirements. The QUEUE command takes on a new life once the system is running. If you type the QUEUE command the system will respond by typing back the current number of free queue blocks in the available queue list. It is by this method that you may keep tabs on the operation of your system as far as queue block usage.

5.2 QUEUE BLOCK USAGE BY THE SYSTEM

This section will list the areas of the monitor which currently make use of the queue system to give you a better idea on how to estimate your particular needs. Remember that this list will probably be expanding in future releases of the monitor. Also add to this any applications that you may write which include the QGET and QRET calls (described later).

The terminal service system makes frequent use of the queue system during output operations. A typical terminal driver may have up to four or five queue blocks in use at any one time for linking buffers and storing immediate data values.

The monitor SLEEP call uses one queue block during the time the job is asleep.

The Persci disk driver uses one queue block while the head is loaded.

The XLOCK AlphaBasic subroutine uses one queue block for each separate system lock that is currently active by any job. This block is not returned to the available list until the lock is released by the job that has it locked.

The line printer spooler, as of version 4.1, uses the queue system to store the printer queue as well as a list of what printers are connected to the system.

Future releases of the file system will make extensive use of the queue system for queued I/O operations and device chains.

5.3 QUEUE SYSTEM MONITOR CALLS

The user may make use of the monitor queue system by using one of the four monitor queue management calls (QGET, QRET, QADD, QINS). These calls are fast for use in interrupt level routines. All calls work through register R3 and no other registers are disturbed. Since most queue blocks will be used in some form of sharable resource chain or interrupt level routine it is required that the processor be locked before executing any of the queue management calls. Not following this rule could result in destruction of the available queue list or inter-job foulups. None of the calls require any arguments to be passed except for the address in R3.

5.3.1 QGET - Obtain a Free Queue Block

This call obtains the first free queue block from the available list and returns its base address in R3. The queue block is first removed from the available list and then all words in the block are cleared to zeros.

5.3.2 QRET - Return a Queue Block

This call returns a queue block to the available queue list in the monitor. The address which was in the first word of the block (usually a link to the next block in your chain) will be returned in R3 after the block has been linked back into the available queue list. All queue blocks that have been allocated by QGET, QADD or QINS should eventually be returned to the monitor by the QRET call when they are no longer needed.

5.3.3 QADD, QINS - Manipulating Queue Blocks

These two calls obtain the first free queue block from the available list similar to the QGET call. The queue block is then linked into your own specific list whose address is in R3. This is because most system calls use queue blocks as elements of some specific list depending on the application. The XLOCK subroutine, for instance, maintains a list of all active system locks and adds or deletes queue blocks from this list as locks are set and reset.

The standard format of these individual lists follows the format of the free list. Each block links to its successor by storing its address in the first word of the block. All other words in the queue block are available for the storage of specific data. The last block in the list will contain a zero in word 1 to mark the end of the list. The QADD call will scan down the chain marked by the address in R3 and then insert the new queue block at the end of the existing list. The QINS call will insert the new queue block in the chain at the point indexed by R3 and link the remaining list elements (if

any) to the newly inserted block. Both calls will then return the address of the second word of the new queue block in R3. This is the base of the data area of the queue block where you may store the data.

Remember that the current size of each queue block is eight words in length. The QADD and QINS calls place a link in the first word so that leaves seven words of data storage for your application. The QRET call always requires the address of the first word when returning the queue block to the available list regardless of the call used to obtain the block.

CHAPTER 6

THE FILE SERVICE SYSTEM

The AMOS monitor has a simple yet powerful device independent file service system which relieves the programmer of the task of I/O coding for each device he wishes to have his program interface with. In addition to this device independence, the monitor contains all routines to manage the disk file system on a logical call basis. The programmer need not be concerned with the exact physical placement of files on the disk except in rare instances where the system software is being developed or tested. The monitor also contains an efficient means for the development of new device drivers to be incorporated into the system when unsupported devices must be interfaced. This section will give a general overview of the file service system and describe the Dataset Driver Block (abbreviated as DDB) which is the descriptor link for all I/O and file calls to the monitor.

6.1 THE DATASET DRIVER BLOCK

All I/O operations and file operations are accomplished by monitor calls with reference to a DDB which defines the device or file being operated upon. Whether the operation is to a unit record device such as a printer, or to a specific file within a file structured device such as a disk, will depend upon the parameters passed to the monitor through the referenced DDB. There is no limit to the number of devices or files that may be active at any given time but there must be one separate DDB for each device or file in use concurrently. There are no internal channel numbers or device numbers to limit the number of concurrently active devices or files. The general sequence of events for the complete processing of a device or file operation can be summed up as follows:

1. The DDB is set up with the defining parameters such as device name, drive number, filename and extension, project-programmer number, etc. This data normally comes from the processing of an ASCII file specification such as DSK1:FILTST.MAC[101,1] by an FSPEC call.

2. The I/O buffers are allocated either directly by the user program or by an INIT call referencing the DDB in use.

3. The logical opening processes for the device or file are performed which normally consists of an OPEN call referencing the DDB.

4. Data transfers to or from the device are performed by either READ and WRITE calls for physical transfers or INPUT and OUTPUT calls for logical transfers.

5. The logical closing processes for the device or file are performed which normally consists of a CLOSE call referencing the DDB.

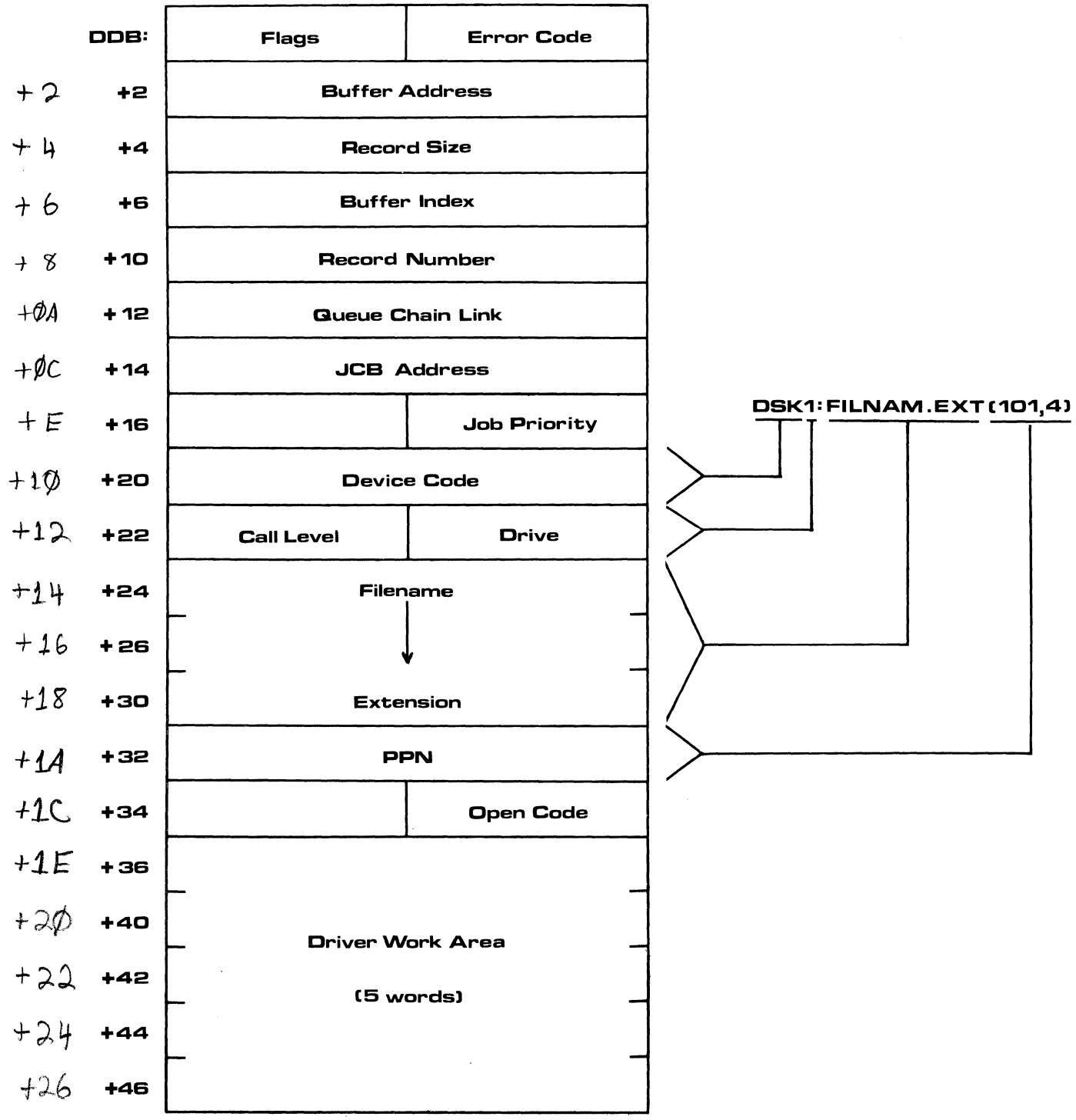
The monitor contains complete error processing routines which allow the programmer to specify (by flags in word 1 of the DDB) whether or not any uncorrectable errors are to result in an automatic error message to the operator on his terminal, an aborting of the program and return to monitor, or both. The programmer may also elect to process the errors himself by checking the error code returned in word 1 of the DDB.

6.1.1 DDB Format

Figure 6-1 shows the format of the DDB which must be allocated within the user program area and set up by the user before any I/O operations can take place. The DDB is 24 (octal) words in size and is usually allocated by a BLKW 24 statement. The DDB can be assigned any tag which will then become the reference tag for all subsequent operations to that dataset. Some of the items in the DDB must be set up by the user before certain operations may be called for while other items are set up and used by the monitor file service routines. The item descriptions that follow will explain the use of each of these.

6.1.1.1 Error Code - This byte is set to a non-zero code at the completion of an I/O operation that was unsuccessful for various reasons. A zero indicates the operation was successful. The user need test this byte only if the error control flag in the flags byte (DDB+1) specifies returning to

the user on an error condition or if the operation allowed a non-fatal error condition to occur. The error codes are listed at the end of this section.



Dataset Driver Block

Fig 6-1

6.1.1.2 Flags - This byte is used to control the flow of the I/O operation and the handling of error codes by the file service routines. The following functions are controlled by the eight flag bits:

0 - set by user to force a return on error condition (abort if clear)
1 - set by user to bypass printing of error messages on error conditions
2 - real-time transfer flag (currently not implemented)
3 - spare
4 - transfer initiated (for internal file service use only)
5 - read if 0 or write if 1 (for internal file service use only)
6 - device INITed - set by INIT call or user if explicit buffer in use
7 - dataset busy (transfer initiated or queued)

6.1.1.3 Buffer Address - This is the 16-bit absolute address of the base of the buffer to be used for all dataset transfers (read and write). It is set by the INIT call which allocates a buffer, or by the user program if it is allocating its own buffer and not using the INIT call. This address is used in conjunction with the flag bit 6 above which indicates that a buffer has been allocated either by the INIT call or by the user. No transfers can take place without a buffer.

6.1.1.4 Record Size - This is the size in bytes for the physical transfer to use. The READ call will transfer this number of bytes from the device to the user buffer beginning with the address in DDB+2. The WRITE call will transfer this number of bytes from the user buffer to the user device. This size is set to the standard buffer size by the INIT call or by the user if he is doing his own buffering. It may be modified by the user for transferring records of variable sizes as long as it does not exceed the buffer size of the capacity of the device or driver in use. Various logical file service routines set this size word during processing such as the OPEN call for the disk which must perform directory operations on a 512-byte buffer at all times.

6.1.1.5 Buffer Index - This is a byte counter which is used by logical routines (INPUT and OUTPUT calls) for keeping track of bytes transferred into and out of the user buffer. Various calls will reset this value and the user will then make use of it and increment it as bytes are transferred into and out of the buffer. Details will be given in later sections where the calls themselves are described. This buffer index word is normally not a true buffer pointer but rather an offset from the buffer base (per DDB+2) to the current byte being manipulated.

6.1.1.6 Record Number - Set by the user to read or write a specific random record from a random access device such as disk. The first record on the device is considered record zero and the record numbers increment sequentially from there. This record number is actually used only by the physical driver routines for READ and WRITE calls but other logical calls set this word to perform transfers to specific disk areas such as directory operations on disk. Most non-disk devices are not random access and therefore this record number will be ignored by the respective drivers.

6.1.1.7 Queue Chain Link - This word is for internal use only and is the link used by the I/O queueing routines for interrupt driven transfers. The user should not alter this word.

6.1.1.8 JCB Address - File service routines store the address of the controlling job's JCB so that interrupt driven drivers can locate the corresponding job for activation on transfer complete status. This word is also for internal use only.

6.1.1.9 Job Priority - Job priority - the current software job priority is set here by file service routines to specify the priority of the transfer in queued operations. This byte is for internal use only. The top byte of this word (DDB+17) is currently not used.

6.1.1.10 Device Code - The three character device code (packed RAD50) must be set here by an FSPEC call or directly by the user before any IO operations may be performed.

6.1.1.11 Drive - Used only by drivers for devices with multiple drives, this byte must be set to specify the drive to be used for the transfer. A -1 byte (octal 377) may be used to indicate the current default drive number. If the device is DSK the default drive used will be the drive that the current user is logged in under. Other devices may have different defaults.

6.1.1.12 Call Level - For internal use only, this byte is used to keep track of the level of nesting of the file service calls for proper error recovery handling. This byte must be zero before the first file call is executed.

6.1.1.13 Filename and Extension - Three words which contain the RAD50 packed filename and extension for file structured devices. These words are ignored by drivers for devices which are not file structured but they may cause funny error messages if they are not set to zero values.

6.1.1.14 PPN - This is the octal project-programmer bytes for the area to be used to locate the file. Used only on file structured devices which are multi-user based such as disk. A zero causes the default value to be the current PPN which the job is logged in under. To prevent strange and erroneous error messages, this word should be zero, if not used.

6.1.1.15 Open Code - This byte is set by the OPEN call to indicate the mode of the open statement for future processing operations. It is normally ignored by drivers for devices which are not file structured. It is for internal use only and should not be modified by the user. The corresponding top byte of the word (DDB+35) is currently not used. The following open codes are in use:

0 - file is not open
1 - file is open for sequential input (OPENI call)
2 - file is open for sequential output (OPENO call)
6 - file is open for random input/output (OPENR call)

6.1.1.16 Driver Work Area - The remaining five words are for internal use by the device drivers for links, record counts, etc and should not be modified by the user during processing. Not all drivers make use of the work area but it must be there if device independence is to be preserved.

6.1.2 Device Transfer Buffers

Each dataset must have an associated transfer buffer for input and output operations to take place through. This buffer must be allocated either directly or through use of the INIT call which allocates the buffer as a

memory module by using a GETMEM call. The INIT call will allocate a standard size buffer for the device being used (the size of the buffer is defined within the driver itself). If you do not wish to use the INIT call

you may allocate any size buffer you wish (must be large enough for any logical calls to be performed) and then set its address in DDB+2. Refer to the section detailing the I/O calls themselves for more details on the use of these buffers.

6.1.3 Error Handling

When an error occurs during any file service call the file service routines will normally perform typical error correction procedures. If the error is fatal (uncorrectable) two operations may or may not take place depending on the setting of bits 0 and 1 in the flags byte at DDB+1. First, bit 1 is tested and if it is not set, the monitor outputs a standard error message to the user terminal giving the type of call that failed, the file specification for the device that the error occurred on, and the reason for the error. The appropriate error code is also placed in the error byte at DDB+0 for later testing by the user. Second, bit 0 of the flags byte is tested and if it is not set, the user program is aborted by the file service system and a return to monitor mode is made. The user will normally set these bits on before any I/O calls are made if it is desired to process the errors within the user program itself.

6.1.3.1 Error Codes - The following list gives the error code (in octal) returned in the DDB error byte by the file service system along with the reason for the error:

01 - file specification error (FSPEC)
02 - insufficient free memory for buffer allocation (INIT)
03 - file not found (OPENI, OPENR, DELETE, RENAME)
04 - file already exists (OPENO)
05 - device not ready (all calls)
06 - device full (OUTPUT)
07 - device error (all calls)
10 - device in use (ASSIGN)
11 - illegal user code (all file calls)
12 - protection violation (OPENO, OPENR, DELETE, RENAME)
13 - write protected (all output calls)
14 - file type mismatch
15 - device does not exist (all calls)
16 - illegal block number (READ, WRITE)
17 - buffer not inited (all calls except INIT)
20 - file not open (READ, WRITE, INPUT, OUTPUT, CLOSE)
21 - file already open (all OPEN calls)
22 - bitmap kaput (all disk bitmap calls)
23 - device not mounted (all calls)
24 - invalid filename (OPENO, FSPEC, DSKCTG)

At the conclusion of every file service monitor call the error byte at the

base of the DDB is tested for the convenience of the user program. This allows you to test for an error status directly after the call with a BNE instruction without having to first explicitly test the byte with a TSTB instruction. This, of course, will only apply if you have the error trapping bit set in the DDB status word to prevent the job from being aborted on a file error.

6.2 FILE SERVICE MONITOR CALLS

This section will describe the file service calls which are available to the user program for both logical and physical I/O operations. All calls have the same general format which uses a single argument representing the dataset driver block (DDB) to be used for the operation. See the preceding chapter for a complete description of the DDB format. In brief, the calls to be described in this section are:

FSPEC	process a device specification
INIT	initialize a dataset driver block buffer
LOOKUP	lookup a file to see if it exists
OPENI	open a file for sequential input
OPENO	open a file for sequential output
OPENR	open a file for random input/output
CLOSE	close a file to further processing
READ	read a physical record
WRITE	write a physical record
INPUT	read a logical record
OUTPUT	write a logical record
WAIT	wait for an I/O operation to finish
DELETE	delete a file
RENAME	rename a file
ASSIGN	assign a device to a job
DEASGN	deassign a device from a job

6.2.1 FSPEC - Process an ASCII Filespec

The FSPEC call is used to process an ASCII file specification from a command line (or any other ASCII buffer) and set up the parameters in the DDB according to the results of the processing. The ASCII file specification must be indexed by R2 and must be in the standard format of dev:filnam.ext[ppn] with a valid termination character if a short default specification is used.

The FSPEC call is slightly different from the rest of the I/O calls in that it allows a second argument to be used if desired. This argument must be the default extension for the filename parameter to be used in the event that the file specification does not contain an explicit extension (identified by a period after the filename). If the second argument does

not exist the FSPEC processor will not process the input file specification past the colon which terminates the device/drive parameters.

The device code (3 characters) will be packed RAD50 and stored in DDB+20 if it exists as marked by the terminating colon. The drive number will be stored in the byte at DDB+22 if it exists. If the device code does not exist, the current default device (stored in the job's JCB item JOBDEV) will be stored in DDB+20. If the drive number is not in the input spec an octal 377 will be stored in DDB+22 to flag the default drive number to the device driver.

The filename and extension are then processed unless no second argument was used in the call in which case the FSPEC processor returns to the user at this point. The filename and extension are packed RAD50 and stored in the three words at DDB+24 through DDB+30. If no filename is entered in the input specification the word at DDB+24 will be cleared to zero to flag the absence of the filename parameter. If a filename is entered but no extension is entered then the default extension specified in the second argument of the FSPEC call is stored as the extension in DDB+30.

If a project-programmer number is in the file specification (marked by a left square bracket "[") it will be processed and stored in DDB+32. If no ppn is entered, DDB+32 will be cleared to zero to flag its absence.

At the conclusion of the processing of the input file specification, the index R2 will be pointing to the termination character (the first character following the file specification string). If an error in the input string is detected the FILE SPECIFICATION ERROR message will be printed (unless suppressed by bit 1 in DDB+1) and the program will be aborted (unless suppressed by bit 0 in DDB+1). The error code 01 will be set in DDB+0 error code byte.

No other modifications take place to the DDB area except that the error byte at DDB+0 is cleared at the start of the FSPEC processing. If the user does not use the FSPEC call to set up his DDB then he must use some other form of explicit code to insure that the DDB is set up properly to define the device and file for any subsequent I/O operations.

6.2.2 INIT - Initialize the DDB

The INIT call is the normal means for allocating the dataset buffer and initializing the DDB for processing. The INIT call will locate the device driver (searching area 1,6 on DSK0 if not in memory) and then allocate a standard size buffer based on the size specified in the driver. Bit 6 of the flag byte at DDB+1 will be set to indicate the initialization. The address of the buffer will be set into DDB+2 and the size in bytes will be set into DDB+4.

There are no calls which deallocate the buffer once it has been allocated by

the INIT call. Multiple OPEN-CLOSE processes may be performed on the DDB once the INIT has been done. The buffer is temporary and will be deallocated automatically when the program exits to monitor or it can be explicitly deallocated by using the DELMEM call with the address stored in DDB+2. Recall that the buffer is allocated as a standard memory module with a GETMEM call.

NOTE

All file service calls with the exception of the FSPEC call require the use of a disk buffer and therefore must be preceded by the INIT call for processing.

6.2.3 LOOKUP - Find the File

This is a form of the OPEN call which does nothing except search for the file and return an error code if it is not found. The file is not actually opened for processing and an OPENI call must be used if the file is to be subsequently read from. The LOOKUP call is useful for determining if a file that is about to be opened for output already exists so that it can first be deleted by the DELETE call. The LOOKUP call is ignored for devices which are not file structured.

The LOOKUP call is also useful for some system programming techniques since it returns parameters about the file in the DDB work area. The work area is located in the last five words of the DDB. The first three words of this work area are loaded with the three words of the directory item if the file is found. These three words are the number of records in the file, the number of active data bytes in the last record, and the record number of the first data record in the file. Refer to the appendix titled "Disk Structure Format" for complete details on the directory format.

6.2.4 OPENI - Open a File for Input

The OPENI call locates a file in a file structured device and sets up the DDB parameters (work area) for subsequent INPUT processing. An error results if the file is not found. The code 01 is set into DDB+34 to flag the OPENI operation. The OPENI call is normally followed by a series of INPUT calls which deliver sequential records from the file to the user buffer. The OPENI call is ignored for devices which are not file structured.

6.2.5 OPENO - Open a File for Output

The OPENO call first searches the specified device in the specified user area and returns an error if the file already exists. If it does not, the DDB is set up for OUTPUT processing. The code 02 is set into DDB+34 to flag the OPENO operation. The OPENO call is normally followed by a series of OUTPUT calls which transfer data from the user buffer to sequential records in the file. The OPENO call is ignored for devices which are not file structured.

6.2.6 OPENR - Open a File for Random Processing

The OPENR executes basically the same as the OPENI call but the code stored in DDB+34 is 04 to flag random processing. The file located for random processing must be a contiguous file. The OPENR call is normally followed by a series of INPUT and OUTPUT calls which transfer data between specific records in the file and the user buffer in both directions. The OPENR call is also ignored for devices which are not file structured.

6.2.7 CLOSE - Close a File

The CLOSE call finishes up logical processing of a file and clears the open code in DDB+34. No further INPUT or OUTPUT operation may occur once a file has been closed. No action is normally done on a file which is open for input. For files open for output, the final record is written out and the file is added to the directory system on the specific device. The CLOSE call is ignored for devices which are not file structured.

6.2.8 READ - Perform a Physical Transfer

This is the physical transfer call for reading input data from a device. No check is made for file open status since the READ call is not a logical file call.

6.2.8.1 Sequential Devices - For sequential access devices such as a paper tape reader, the READ call will deliver one record from the device to the user buffer. The size of this record will normally be the number of bytes specified in DDB+4 but this may not necessarily be true if the driver does not transfer under the rules of the system. If the device is not capable of generating the requested number of bytes per DDB+4 (such as a tape reader which runs out of tape) a lesser number may be transferred in which case the count in DDB+4 will be adjusted to reflect the true number actually transferred to the user buffer.

6.2.8.2 Random Devices - For random access devices such as disk, the user must specify the record number to be located and read by placing that number into DDB+10 before executing the READ call. Most random access devices will always transfer the requested number of bytes per DDB+4 into the user buffer. An error will result if the record number is not within the range of the specific device. For example, the standard AMOS floppy disk is structured as 500 (decimal) records of 512 bytes each. The legal record numbers therefore range from 0 through 499 decimal. Similar range restrictions apply for each random device.

6.2.8.3 Interrupt Structure - The system allows interrupt driven devices to be queued and processed in a priority fashion. Normally, the execution of a READ call will suspend the running of the user program until the transfer has been completed at which time the user job will be reactivated. The user may optionally set the realtime bit (bit 2) in the flag byte at DDB+1 to force an immediate return to the program once the transfer has been queued or initiated. The user must then either test the dataset busy bit (bit 7) of the flag byte or use the WAIT call to stall until the transfer has been completed. The dataset busy flag will be reset when the transfer has been completed. The user must also then check for errors. The realtime bit is ignored for devices which are not interrupt driven or whose drivers do not run under the I/O queue system.

6.2.9 WRITE - Perform a Physical Write

This is the physical transfer call for writing data to a device. No check is made for file open status since the WRITE call is not a logical file call.

6.2.9.1 Sequential Devices - For sequential access devices such as a printer, the WRITE call will deliver one record to the device from the user buffer. The size of this record will be the number of bytes specified in DDB+4. The driver is responsible for the correct transfer count and the user may alter the number in DDB+4 for each new WRITE call to the same device for the writing of variable length records.

6.2.9.2 Random Devices - For random access devices such as disk, the user must specify the record number to be located and read by placing that number into DDB+10 before executing the WRITE call. Most random access devices will always transfer the requested number of bytes per DDB+4 into the user buffer. An error will result if the record number is not within the range

of the specific device. The standard AMOS floppy disk is structured as 500 (decimal) records of 512 bytes each. The legal record numbers therefore range from 0 through 499 decimal.

6.2.9.3 Interrupt Structure - The system allows interrupt driven devices to be queued and processed in a priority fashion. Normally, the execution of a WRITE call will suspend the running of the user program until the transfer has been completed at which time the user job will be reactivated. The user may optionally set the realtime bit (bit 2) in the flag byte at DDB+1 to force an immediate return to the program once the transfer has been queued or initiated. The user must then either test the dataset busy bit (bit 7) of the flag byte or use the WAIT call to stall until the transfer has been completed. The dataset busy flag will be reset when the transfer has been completed. The user must also then check for errors. The realtime bit is ignored for devices which are not interrupt driven or whose drivers do not run under the I/O queue system.

6.2.10 INPUT - Perform a Logical Read

The INPUT call is the logical equivalent of the READ call for logical processing of datasets. The INPUT call reads a logical record within a file or device dataset under the control of the specific driver in use. A dataset must be opened for input (OPENI) or random access (OPENR) before INPUT calls are performed. The INPUT call first sets the standard buffer size into DDB+4 so the user may not use this call to transfer non-standard record sizes. The number of bytes actually read may be less than the standard record size due to the driver processing or due to an end of file condition. The actual number of bytes transferred will be set into DDB+4 by the driver routine.

6.2.10.1 Sequential File Processing - The main use of the INPUT call is in logical sequential file processing and the INPUT call therefore sets up the buffer index value in DDB+6 to direct the processing of the data by the user routines. This index value is actually the offset to the first byte of valid data within the user buffer whose base address is at DDB+2. For unit record devices this value will be zero since all data within the buffer is user data. For sequential disk files, however, the first word in each record within the file is a link word to the next record and therefore the value set into DDB+6 by the disk driver will be 2 so that processing starts with the third byte in the user buffer.

6.2.10.1.1 Example - The following subroutine is normally used to get each byte of data from a sequential file:

```
;Subroutine to get next byte from file defined as INDDB and leave it in R1
;
INBYTE: CMP     INDDB+6,INDDB+4 ;is the buffer empty?
        BLO     INBG            ;  no - get next byte
        INPUT   INDDB           ;read next logical record into buffer
        TST     INDDB+4         ;check for end of file (no data transferred)
        BEQ     INEOF           ;  go to end of file routine
INBG:   PUSH    INDDB+2         ;stack the buffer base address
        ADD     INDDB+6,@SP     ;  and add the index offset to get position
        MOVB    @(SP)+,R1       ;pick up the next byte from user buffer
        AND     #377,R1         ;insure upper byte is cleared in R1
        INC     INDDB+6         ;increment the buffer index for next time
        RTN                     ;subroutine return
```

6.2.10.2 Random File Processing - A special situation is involved for files opened for random access by the OPENR call. Instead of reading the next sequential record, the specific relative record whose number is in DDB+10 will be read into the user buffer. The user first sets this number up and then executes the INPUT call. The record number is actually relative to the base of the file and has no direct relationship to the physical record on the device as would be returned by a READ call.

6.2.10.3 Special Devices - For devices that do not implement special processing of logical calls the INPUT call performs a READ call instead.

6.2.11 OUTPUT - Perform a Logical Write

The OUTPUT call is the logical equivalent of the WRITE call for logical processing of datasets. The OUTPUT call writes a logical record to a file or device dataset under the control of the specific driver in use. A dataset must be opened for output (OPENO) or random access (OPENR) before OUTPUT calls are performed. The OUTPUT call will transfer the number of bytes in DDB+4 but it will normally do it as a standard record (depends on the driver in use). The user is discouraged from attempting to use the OUTPUT call for transferring non-standard record sizes.

6.2.11.1 Sequential File Processing - The main use of the OUTPUT call is in logical sequential file processing and the OUTPUT call therefore sets up the buffer index value in DDB+6 to direct the processing of the data by the user routines. This index value is actually the offset to the first byte position for valid data within the user buffer whose base address is at DDB+2. For unit record devices this value will be zero since all data within the buffer is user data. For sequential disk files, however, the first word in each record within the file is a link word to the next record and therefore the value set into DDB+6 by the disk driver will be 2 so that processing starts with the third byte in the user buffer.

6.2.11.1.1 Example - The following subroutine is normally used to put each byte of data to a sequential file:

```
;Subroutine to put next byte from R1 into file defined as OTDDB
;
OUTBYT: PUSH    OTDDB+2          ;stack the buffer base address
        ADD     OTDDB+6,@SP      ;  and add index offset to get position
        MOVB    R1,@(SP)+        ;move data byte to user buffer
        INC     OTDDB+6          ;increment the buffer index offset value
        CMP     OTDDB+6,OTDDB+4  ;is the user buffer full now?
        BLO     OTBX             ;  not yet so return
        OUTPUT  OTDDB            ;output the current logical record to file
OTBX:   RTN                      ;subroutine return
```

6.2.11.2 Random File Processing - A special situation is involved for files opened for random access by the OPENR call. Instead of writing the next sequential record, the specific relative record whose number is in DDB+10 will be written out from the user buffer. The user first sets this number up and then executes the OUPUT call. The record number is actually relative to the base of the file and has no direct relationship to the physical record on the device as would be written by a WRITE call.

6.2.11.3 Special Devices - For devices that do not implement special processing of logical calls the OUTPUT call performs a WRITE call instead.

6.2.12 WAIT - Wait for IO Completion

The WAIT call is used only for interrupt device processing to stall until the record transfer has been completed. The WAIT call has no effect unless the user has set the realtime flag bit (bit 2) in DDB+2 and the device is interrupt driven. The job will be suspended until the specified dataset is not busy as marked by the flag bit 7 becoming zero. If the dataset is not

busy when the call is executed, an immediate return is made. The WAIT call also checks for an error condition that may have occurred during the transfer process and performs error handling under control of flag bits 0 and 1.

NOTE

At the release of 4.1 no drivers fully support interrupt driven and real time calls. The WAIT call would merely be ignored in these cases and is detailed here in preparation for future releases.

6.2.13 DELETE - Delete a File

The DELETE call deletes a specific file from a file structured device. The filename, extension and ppn (if used) must be set in the DDB before executing the call. An error results if the file is not found. The DELETE call is ignored for devices which are not file structured.

6.2.14 RENAME - Rename a File

The RENAME call renames a specific file on a file structured device. The filename, extension and ppn (if used) must be set in the DDB before executing the call. The new filename and extension must be packed RAD50 into the three words immediately following the DDB in memory. The RENAME call merely locates the directory item for the file and replaces the three words which store the filename and extension. The RENAME call is ignored for devices which are not file structured.

6.2.15 ASSIGN - Assign a Device

The ASSIGN call is used to assign a non-sharable device (such as a printer) to the current user's job by setting a flag in the device's entry in the device table in monitor memory. Once a device has been assigned by this call any attempt to assign it by another job will result in an error. The device will stay assigned to this job until deassigned by the DEASGN call. The ASSIGN call performs no action if the specified device is sharable such as a disk.

6.2.16 DEASGN - Deassign a Device

The DEASGN call is used to deassign a device which has been assigned to the user's job by the ASSIGN call. Once deassigned, the device becomes available for assignment by other jobs. The DEASGN call performs no action if the specified device is sharable of if it is not currently assigned to the user's job. All devices are deassigned when the program exits to the monitor.

6.3 DISK SERVICE MONITOR CALLS

In the previous section we covered the file-oriented monitor calls. Those calls allow you to access data files without regard to the actual structure of the data on the device. Internally, of course, AMOS does have to deal with the structure of the data. This section deals with the monitor calls used to manipulate that structure. A description of the data structures used to maintain files on a device can be found in Appendix A - Disk Structure Format.

The disk presents special problems which require the use of special monitor calls to control the accessing of the directory and bitmap records. These records have a non-sharable attribute associated with them even though the disk in general is a sharable device. For instance, two user programs may not both be updating the same directory records at the same time. The same holds true for the bitmap records. The following monitor calls are used to control the access to these non-sharable records:

DSKCTG - allocates a contiguous file for random processing
DSKALC - allocates the next available record on disk
DSKDEA - deallocates a specific record on disk
DSKBMR - reads disk bitmap and sets reentrant lock flag
DSKBMW - rewrites disk bitmap after user modification
DSKDRL - sets reentrant directory lock for a specific user
DSKDRU - clears reentrant directory lock for a specific user

The access to these records is normally done by the monitor routines as a direct result of normal I/O processing by file service calls. It is a somewhat tricky process and the disk calls should not be used except with extreme caution since misuse could violate the integrity of the file structure on the disk. The following descriptions are directed at those system programmers who are familiar with shared file techniques.

6.3.1 Calling Sequence

All calls use a standard argument which is the address of the associated DDB to be used for the call. In addition to the first argument which is the DDB, some calls use a second argument for processing. The second argument, if used, will be detailed in the description of the call.

6.3.2 The Bitmap Area

The bitmap area is an area in monitor memory which is allocated by the BITMAP program run at system startup time by the BITMAP command in the system initialization command file. This area consists of a status word, a DDB for bitmap reads and writes, and a buffer for the actual bitmap including the hash total words. The format of the bitmap area is as follows:

```
WORD    0               ;Bitmap status word
BLKW    12              ;Partial DDB for bitmap I/O
BLKW    Bitmap-size     ;Bitmap buffer (size depends on device)
BLKW    2               ;Hash total words
```

The device table entry for each drive has the address of the corresponding bitmap area to be used for that drive. More than one drive may share the same bitmap area which will force a rewrite each time a different drive is referenced. This is not efficient timewise but can save some memory for larger devices where the bitmap buffer may be several hundred words or more.

6.3.2.1 The Status Word - The status word (first word in bitmap area) contains two flags which are used to control bitmap access. Bit 0 is the bitmap lock flag and is set to flag that the bitmap is locked and being read or modified by some user job. The DSKBMR call sets this flag on and it is up to the user to clear it after he has finished the bitmap access and modification. Bit 1 is the bitmap rewrite flag which is set to indicate that one or more modifications have been made to the bitmap in memory and that it must be rewritten to disk before being discarded. If the user program modifies the bitmap in memory it must set the rewrite flag to insure that the bitmap is rewritten.

6.3.2.2 The Bitmap DDB - The bitmap DDB is a partial DDB due to the fact that no files are ever referenced and the rest of the DDB is not needed. The bitmap is normally allocated as record 2 of each disk and it extends across successive records for those devices which overflow one record.

6.3.2.3 The Bitmap Buffer - The bitmap buffer area is the exact size required to contain the entire bitmap from the disk. Two extra words are allocated to contain the hash total which is used to insure the integrity of the bitmap in memory and on disk. Each time the bitmap is read or before the bitmap is rewritten this hash total is checked and an error results if it is bad. The hash total is merely the double-word binary sum of the entire bitmap buffer. The user must update this hash total each time he modifies the bitmap or else an error will result when it becomes time to rewrite the bitmap to disk.

6.3.2.4 The Bitmap - The bitmap itself contains one bit for each logical record on the disk structure and this bit is off if the record is free and on if the record is in use by anyone including the system structure records themselves. Each word in the bitmap can define up to 16 records. The first word in the bitmap defines records 0 through 17 (octal) with bit 0 defining record 0 and proceeding upward throughout the word. The second word defines records 20 thorugh 27, and so on. To define the 500 decimal records in a standard IBM-compatible AMOS floppy disk we need 32 words (32 times 16 = 512) with the last word not being totally used. The bitmap itself therefore takes up 34 words which includes the two hash total words.

6.3.2.5 Altering the Bitmap - Altering the bitmap is tricky but the sequence recommended is:

1. Read the bitmap using the DSKBMR call
2. Alter the bitmap as necessary (recompute the hash total)
3. Set the rewrite flag (status word bit 1)
4. Clear the bitmap lock (status word bit 0)
5. Rewrite the bitmap using the DSKBMR call
6. Breathe a sign of relief if it worked

6.3.3 DSKCTG - Allocate a Contiguous Area

The DSKCTG call is used to allocate a contigous file on a random access device. A standard argument is used as the second argument which represents the number of records to be allocated in the file. A search will be made to find the first available hole on the disk which will fully contain the requested number of records. These records are marked as in-use on the disk bitmap and a file descriptor item is added to the user directory. The word which gives the number of bytes in the last record will be set negative to flag this file as contiguous to distinguish it from the normal sequential files. A device full error will result if no hole can be found on the disk which is large enough to contain the file.

6.3.4 DSKALC - Allocate a Record

The DSKALC call is used to allocate one record for use by this user as a directory record or as a file record. A standard argument is used as the second argument which represents the word which is to receive the record number of the allocated record. An error will result if there are no free records left on the specified disk. A DSKBMR call is first performed to insure that the current job has access to the bitmap and then the first free record is located and marked in use. The bitmap record is flagged as modified which will cause it to be rewritten at the next DSKBMW call or if it must be swapped out to make room for another bitmap sharing the same area in memory.

6.3.5 DSKDEA - Deallocate a Record

The DSKDEA call is used to deallocate a specific record on a disk and make it immediately available for use by another user (or the same user). A standard argument is used as the second argument which represents the address of the word which contains the record number of the record to be deallocated. No check is made to insure that this record is allocated to either the current user or any other user. A DSKBMR call is first performed to insure that the current job has access to the bitmap and then the specified record's bit is set to zero to indicate that the record is free. The bitmap record is flagged as modified to force a rewrite.

6.3.6 DSKBMR - Read the Bitmap

The DSKBMR call locates the bitmap area in monitor memory for the specified disk and insures that it is not locked by another job. If it is locked, a stall will be made until it is released. It is then locked for this job and a return is made to the user. The address of the bitmap area will be set into the word specified by the second argument in the calling sequence. The second argument is a standard argument in format. Refer to the description of the bitmap area above and note that the second argument receives the address of this area and not the address of the bitmap itself. The user may locate the bitmap itself because its address is in the third word of the bitmap area (second word of the bitmap DDB).

6.3.7 DSKBMW - Write the Bitmap

The DSKBMW call locates the bitmap area in monitor memory for the specified disk and insures that it is not locked by another job. If it is locked, a stall will be made until it is released. It is then locked for this job and

rewritten to disk from memory unless the hash total is bad. After the rewrite is complete both the rewrite and lock flags are cleared and a return is made to the user.

6.3.8 DSKDRL - Lock the Directory

The DSKDRL call locks the directory for the specified drive for modification by the user program. It is used by such file service routines as CLOSE for output files, DELETE and RENAME calls. If the directory is already locked by another job a stall will be made until it is released. The user program or routine must unlock the directory via the DSKDRU call after the modifications have been made.

6.3.9 DSKDRU - Unlock the Directory

The DSKDRU call unlocks the directory for the specified drive after it has been locked by the DSKDRL call for modification. No action will be performed if the directory is not locked by the current job.

CHAPTER 7

TERMINAL SERVICE SYSTEM

The AMOS monitor has several calls which deliver data to and from both the user terminal and other terminals connected to the system. A terminal is defined as an ASCII character-oriented device which is capable of both output and input. This is the formal definition and does not preclude the use of output-cnly devices on terminal designated ports. Also, the system includes software terminals known as "pseudo terminals" which can be used to control jobs that are not actually associated with a hardware interface on a designated port address. The calls listed here normally input from or output to the terminal which is controlling the job that the call is being executed by. Some calls (as specified) will input from or output to another terminal not connected to the current job or to a pseudo terminal controlling another job.

Programs which make use of the standard terminal service calls that communicate with the user terminal can be run in a job controlled by a pseudo terminal without modification. Keyboard input calls and terminal output calls will always go to the controlling terminal regardless of which job they are running in. The user, therefore, need not be concerned with the physical port address or attributes of the terminal which is controlling the job. The monitor routines handle all this automatically.

7.1 TERMINOLOGY

Due to a holdover from older system terminology most terminal output calls reference the device name of "TTY" which used to define the teletype device on systems which normally used teletypes as terminals. The input device of the teletype was then called the keyboard and the calls reference the device name of "KBD." These are strictly mnemonics and hold no true bearing to the attributes of the physical terminals which now are more commonly the higher speed video CRT terminals.

7.2 THE TERMINAL LINE TABLE

Each terminal has associated with it a terminal line table which is a work area in monitor memory set up to contain the parameters and work areas associated with the control of the terminal device. Most of the items in this terminal line table are for internal use only and the user need not be concerned with them. The JOBGET Rx,JOBTRM call may be used to set an index to the associated terminal line table so that the user may inspect or modify the items within.

7.2.1 The Terminal Status Word

The only item that the user normally need be concerned with is the terminal status word which is the first word in the terminal line table. This word has certain flags in it which the user may modify to alter the operation of his terminal calls. The terminal status word has the following flag positions defined:

Bit 0 - user sets to force image mode input (see KBD call)
Bit 1 - user sets to suppress echoing of input characters
Bit 2 - user sets to allow escapes to be processed (as in EDIT)
Bit 4 - user sets to allow lower case input (disables conversion)
Bit 7 - internal flag used to indicate output is in progress
Bit 9 - flag used to indicate "hog" mode for terminal (set by TRMDEF)
Bit 10 - user sets to indicate terminal runs in local mode (no echo)

The terminal status word is cleared each time the user program exits back to monitor mode upon program completion thereby restoring normal terminal operation regardless of program operation.

7.3 THE TERMINAL SERVICE CALLS

AMOS supports the terminals connected to the system with 17 monitor calls to perform both input and output from any of the terminals connected to the system.

7.3.1 KBD - Fetch a Line of Data

The KBD call accepts one full line of input from the user terminal into a monitor line buffer and then sets index R2 to the base of that buffer for the user reference. During the inputting of the line, the user job is set into the terminal input wait state thereby consuming no CPU time until the line is finished. All normal line editing features are active (rubout, control-U, tab, etc) and a control-c input will abort the job unless the user has set up control-c trapping via the JOBICP item in the JCB for the

job. The line will be terminated when a carriage-return or a line-feed is entered. The carriage-return will have a line-feed automatically appended to it by the monitor and a null byte will be set after the line-feed character.

If the echo-suppress flag is set in the terminal status word, normal echoing of the input characters will be suppressed such as when the password is being entered for the LOG command. If the image-mode input flag is set the KBD command takes on a whole new lease on life. No editing is performed and instead of one line being accepted, only one character is accepted and it is delivered back to the user in register R1 instead of setting register R2 to the monitor line buffer. Image mode input echoing is still under control of the echo-suppress flag as in normal line mode.

7.3.2 TTY - Output One Character

The TTY call outputs one character from register R1 to the controlling terminal and then returns. Tabs are echoed as spaces up to the next modulo-8 carriage position unless the image-mode output flag is set in the terminal status word. If the job is running under the command of a control file, the character will only be output to the terminal if the output suppress command is in the normal state (:R revives it, :S silences it).

7.3.3 TIN - Get an Input Character

Gets the next input character from either the terminal input buffer or from the command string if the job is controlled by a command file. The character is delivered in R1. This call is normally only used within the operating system itself and not by user programs.

7.3.4 TOUT - Output One Character

Outputs one character to the controlling terminal of the job or to the job which has this job attached (by the address in the JOBATT item). This call differs from the general TTY call in that the command file status is not checked by the TOUT call. The TOUT call, like the TIN call, is normally only used within the operating system itself.

7.3.5 TAB - Output One Tab

This is a convenience call which outputs a single tab character to the user terminal. It is in effect the same as the code sequence:

```
MOVI    11,R1
TTY
```

7.3.6 CRLF - Output a Carriage-Return / Line-Feed

This is a convenience call which outputs a carriage-return and line-feed pair to the user terminal. It is in effect the same as the code sequence:

```
MOVI    15,R1
TTY
MOVI    12,R1
TTY
```

7.3.7 TTYI - Output a String of Characters

The TTYI call outputs a string of characters which follows the call itself up to but not including a null byte. The call could be used as follows to output two lines of data to the user terminal:

```
TTYI
ASCII   /LINE 1 DATA/
BYTE    15
ASCII   /LINE 2 DATA/
BYTE    15,0
EVEN
```

The TTYI call will also automatically append a line-feed to all carriage-returns which are included in the string.

7.3.8 TTYL - Output a String of Characters Indexed

The TTYL call is similar to the TTYI call in that it outputs a string of ASCII characters up to a null byte. The string of characters for the TTYL call may be anywhere in memory and not inline with the call itself in the program flow. The TTYL call takes one standard argument which is the address of the message to be output. The TTYL call is therefore useful for outputting from a table of messages by setting an index to the specific message within the table (per some numeric director code) and then using that register as the argument to the TTYL call. The TTYL call also appends a line-feed to each carriage-return in the string.

7.3.9 PTYIN - Place Character in Input Buffer

The PTYIN call allows one job to force a character into the input buffer of another job which is probably controlled by a pseudo terminal. The PTYIN call takes two standard arguments. The first argument is the data byte to be sent to the other job and the second argument is the address of the JCB of the job into which the character is to be forced. The PTYIN call is the method by which the FORCE operator command does its dirty work.

7.3.10 PTYOUT - Fetch Character from Output Buffer

The PTYOUT call allows one job to get a character from the terminal output buffer of another job which is controlled by a pseudo terminal. If no output is available from the specified job, the calling job is put to sleep until a character is available. The PTYOUT call takes two standard arguments. The first argument is the address of the byte which will receive the data character and the second argument is the address of the JCB from which the character is to be stolen.

7.3.11 TTYIN - Fetch Another Jobs Input

The TTYIN call allows one job to get waiting input data from the terminal input buffer of another job. This call has not yet been fully inplemented.

7.3.12 TTYOUT - Place a Character in Another Jobs Output

The TTYOUT call allows one job to put data into another jobs terminal output buffer. This call, like the TTYIN call, is not yet fully implemented.

7.3.13 TRMICP - Process Input Character Within Interface Driver

The TRMICP call is executed from within a terminal interface driver to process one character which has just been received from the terminal by the hardware interface. R1 must contain the input character to be processed and R5 must index the terminal definition table entry for the specific terminal being serviced. TRMSER will then take the character and pass it to the terminal driver input routine for pre-processing if desired. When the terminal driver passes it back to TRMSER it will then be edited for control codes and other special characters and then added to the terminal input buffer. All the pertinent flags will be set automatically to indiciate actions to be taken by the application program when it requests the input

data. If the input character is a break character (line-feed) or if image mode is active the associated job will be awakened to process the available data.

7.3.14 TRMOCP - Process Output Character Within Interface Driver

The TRMOCP call is executed from within a terminal interface driver to get the next output character from TRMSER which is to be sent to the terminal. This is usually in response to an interrupt from the interface board indicating that the prior character has been fully output and the board is ready to transmit the next character. R5 must index the terminal definition table entry for the specific terminal being serviced and R1 will get the next available character upon return from TRMSER processing of the call. If there is no more output available in the output buffer, R1 will be set to -1 as a flag and the associated job will be awakened to fill the output buffer again.

7.3.15 TRMBFQ - Process Output Characters Within Terminal Driver

The TRMBFQ call is a physical output call usually executed from within a terminal driver or a monitor routine. There are, however, times when it can be used by an assembly language application program. The TRMBFQ call effectively adds a buffer full of data characters to the output buffering system for a specific terminal. It does this by linking the buffer into the dynamic output queue list used by TRMSER for this terminal. When using this call R2 must index the buffer to be queued, R3 must contain the number of characters in the buffer, and R5 must index the terminal definition table entry for the specific terminal. The TRMBFQ call will perform the output initiation function if the output system for the terminal is currently idle.

7.3.16 TBUF - Output Large Amounts of Data

The TBUF call is the normal call for user programs to use for queueing up large amounts of data into the terminal output system of a terminal where the single character calls are considered inefficient. It is a buffered call in that it works through the two output buffers for the terminal as opposed to going directly into the output queue system. If more data is attempted to be output via the TBUF call then there is currently room for in the output buffers, the user job will be suspended while the output buffers are unloaded to the terminal. Each time one of the output buffers is emptied the job is awakened and the TBUF call proceeds to fill that buffer. This continues until the original amount of data is exhausted whereby the call returns to the user program. When the call is executed R2 must index

the buffer to be output and R3 must contain the number of characters to be output (similar to the TRMBFQ call). R5 need not index the terminal definition table entry since this is a user level call.

7.3.17 TCRT - Call Special Terminal Driver Routines

The TCRT call is the linkage into the special processing routine portion of a terminal driver. R1 usually contains a two-byte code which is interpreted by the terminal driver routine as a special function such as cursor positioning or special editing action. The only action actually performed by the TCRT call within TRMSER is to locate the terminal driver for the attached terminal and call the driver control routine within it. The user must refer to the actual driver listing to determine the action performed relative to the code passed to it in R1.

7.3.17.1 Standard Functions - The TCRT call will most commonly be used for controlling such special CRT functions as cursor addressing and screen clearing. To maintain compatability between terminal drivers, Alpha Micro has defined the following functions within the terminal drivers it supports.

7.3.17.1.1 Cursor Addressing - To perform cursor addressing, R1 is loaded with a two byte argument defining the screen row and column to which the cursor is to be moved. The high-order byte is loaded with the row, and the low-order byte is loaded with the column. The uppermost-leftmost (Home) position is considered to be column 1, row 1.

7.3.17.1.2 Other Functions - To perform other special CRT functions, the high-order byte of R1 should be loaded with 377 (octal). The low-order byte is then loaded with one of the special function codes as listed below.

0	Clear Screen and set normal intensity
1	Cursor Home (move to 1,1)
2	Cursor Return (move to column 1)
3	Cursor Up
4	Cursor Down
5	Cursor Left
6	Cursor Right
7	Lock Keyboard
10	Unlock Keyboard
11	Erase to End of Line
12	Erase to End of Screen
13	Enter Background Display Mode (reduced intensity)
14	Enter Foreground Display Mode (normal intensity)

15	Enable Protected Fields
16	Disable Protected Fields
17	Delete Line
20	Insert Line
21	Delete Character
22	Insert Character
23	Read Cursor Address
24	Read Character at Current Cursor Address
25	Start Blinking Field
26	End Blinking Field
27	Start Line Drawing Mode (enable alternate character set)
30	End Line Drawing Mode (disable alternate character set)
31	Set Horizontal Position
32	Set Vertical Position
33	Set Terminal Attributes

Not all terminal drivers have all of the above functions simply because all terminals do not have all of the functions. If your terminal has additional features, Alpha Micro recommends starting at 100 (octal) when assigning function codes.

7.3.18 Message Calls

There are three calls which have been defined in SYS.MAC as macros using the TTYI call. These calls are for the convenience of the programmer and to make the program more readily understandable. They all take a single argument which is an ASCII message string to be output to the user terminal. Due to the way that macro arguments are processed, if the message has leading or trailing spaces or if it has imbedded commas, it must be enclosed in angle brackets or part of it will be lost. The three calls are:

```
TYPE    msg     ;Types the message on the user terminal as is
TYPESP  msg     ;Types the message and appends one space to it
TYPECR  msg     ;Types the message and appends a CRLF pair to it
```

The macros are defined in SYS.MAC as follows:

```
DEFINE  TYPE    MSG
        TTYI
        ASCII   /MSG/
        BYTE    0
        EVEN
        ENDM

DEFINE  TYPESP  MSG
        TTYI
        ASCII   /MSG' /
        BYTE    0
        EVEN
        ENDM
```

```
DEFINE  TYPECR  MSG
        TTYI
        ASCII   /MSG/
        BYTE    15,0
        EVEN
        ENDM
```

It should be noted that the message may not contain any slashes since these are used as delimiters for the ASCII statement in the macros.

CHAPTER 8

CONVERSION MONITOR CALLS

8.1 NUMERIC CONVERSION CALLS

The AMOS monitor contains two calls which perform conversions from a single binary word value to an ASCII formatted decimal or octal string. Options for the conversion allow the string to be sent to the user terminal, to an output file or to a buffer in memory. Options also allow control of the result format.

8.1.1 Calling Format

Both calls have the same general format and take two arguments, each of which must be an expression that evaluates down to a byte value within the specified range. The two calls are:

```
DCVT    size,flags      ;Convert binary number in R1 to decimal
OCVT    size,flags      ;Convert binary number in R1 to octal
                        ; (hexadecimal if J.HEX is set for this job)
```

8.1.1.1 Size Byte - The size byte determines the number of digits in the output result. A zero size specifies a floating format in which the number of digits used will be just enough to fully contain the result. A non-zero size specifies a fixed number of digits for the result with leading zeros being replaced by blanks. In either form, if the R1 value is zero at least one zero digit will be output as the result.

8.1.1.2 Flags - The flags byte contains six flags which control the destination of the result string and also some other formatting options. The following list gives the flag bit positions and the action taken when the flag is set:

1 - Bit 0 - disables leading zero blanking
2 - Bit 1 - outputs the result to the user terminal
4 - Bit 2 - outputs the result to the file whose DDB is indexed by R2
10 - Bit 3 - puts result in memory at buffer indexed by R2 and updates R2
20 - Bit 4 - adds one leading space to the result
40 - Bit 5 - adds one trailing space to the result

Note that the maximum value which can be displayed using these calls is the maximum value of a 16-bit word. All numbers are considered unsigned so the largest decimal number is 65535, the largest octal number is 177777, and the largest hex number is FFFF.

If the size byte is non-zero, the sense of the leading zero blanking flag described below is reversed. In other words, when the size byte is zero, the conversion calls default to leading zero blanking, with bit 0 turning that blanking off. When the size byte is non-zero, the calls default to leading zeroes, with bit 0 specifying that leading zeroes are to be blanked.

The following examples may clarify things a bit. All examples assume the value in R1 is 964 (decimal) and the letter "b" in the result field indicates a blank.

```
DCVT    0,2     prints 964
DCVT    0,22    prints b964
DCVT    0,42    prints 964b
DCVT    5,2     prints 00964
DCVT    5,3     prints bb964
DCVT    5,43    prints bb964b
DCVT    5,62    prints b00964b
DCVT    2,2     prints 64 (the 9 is lost)
```

8.2 RAD50 CONVERSION MONITOR CALLS

Radix-50 packing is used throughout the system where the packing of filenames and other data entities lends itself. Radix-50 (RAD50) packing is a system by which 3 ASCII characters may be packed into a single 16-bit word using a special algorithm based on the value of octal 50. The character set that may be packed RAD50 is limited in scope to the alphanumeric characters, the period, the dollar sign, and the blank. The following list gives the legal characters that may be packed RAD50 and their equivalent octal codes:

Character	RAD50 code
blank	0
A-Z	1-32
$	33
.	34
0-9	36-47

There is no character for the RAD50 code 35.

8.2.1 RAD50 Packing Algorithm

The packing algorithm for a three-character input to a 16-bit RAD50 result is:

1. The first character code is multiplied by 3100 octal (50x50)
2. The second character code is multiplied by 50 and added to the first
3. The third character code is added to the above to form the result

The unpacking algorithm merely reverses the above sequence to get the triplet.

8.2.2 Packing and Unpacking Calls

There are two monitor calls which perform the above packing and unpacking algorithms. Both calls use registers R1 and R2 as indexes to the components and require no calling arguments.

8.2.2.1 PACK - Pack Three ASCII Characters into RAD50 - The triplet (3 ASCII characters) indexed by R2 is packed into RAD50 form and the result is left in the word indexed by R1. R1 is incremented by 2 to receive the next result word for multiple packing. R2 is left indexing the first character which was not included in the packing of this triplet. The PACK call will terminate packing and force blank fill for any input which does not contain 3 valid RAD50 characters. For the PACK call, a blank will be considered an illegal input character and will terminate packing.

8.2.2.2 UNPACK - Unpack Three RAD50 Characters into ASCII - The word in the address indexed by R1 will be unpacked and the triplet will be left in the three bytes beginning with the byte currently indexed by R2. R1 will be incremented by 2 for the next word and R2 will be incremented by 3 for the next triplet result. Blanks are legal in unpacking and will be placed into the result if they are decoded from the input word.

8.3 PRINTING CONVERSION CALLS

There are three calls in the monitor which accept a system unit input and convert the unit to standard printable form and then output it to the user terminal. These calls are used to print out file specifications, filenames, and project-programmer numbers. Each call takes one standard argument which addresses the system unit to be converted and printed.

8.3.1 PFILE - Output a Filespec From a DDB

The argument addresses a file DDB and the PFILE call extracts the parameters in the file specification words and prints them in the standard format of dev:filnam.ext[ppn] on the user terminal.

8.3.2 PRNAM - Output a Filename

The argument addresses a 3-word filename.extension block (packed RAD50) and the PRNAM call prints the converted result in the standard format of filnam.ext on the user terminal.

8.3.3 PRPPN - Output a PPN

The argument addresses a 1-word project-programmer code and the PRPPN call prints the converted result in the standard format of proj,prog on the user terminal. The PPN will be output in octal regardless of the setting of J.HEX.

CHAPTER 9

INPUT LINE PROCESSING CALLS

When a program is executed by an operator command, register R2 is left pointing to the first non-blank character on the command line which follows the command name itself. The remainder of the line is normally interpreted by the particular program and used to determine the files to be acted on, the record number to be dumped, the devices to be accessed, etc. For example, the MACRO call requires the name of the program and any switch options to follow the MACRO command name on the same line. The macro assembly program then processes the program name and the switch options by way of the R2 index which was left indexing the rest of the command line. This command line is actually the user's terminal input buffer.

In addition to the command input line, the KBD monitor call also leaves R2 set to the input line buffer which contains the user input data. Also, various translators and file processing programs may read in a line of data and then set index R2 to the base of that line for scanning. For this reason, there exist a number of monitor calls which perform scanning and conversion functions based on an input line which is indexed by R2. Some of the calls merely test the character indexed by R2 for a specific condition and return with flags set based on the result of the test. In these instances R2 is not modified. In calls which perform scan conversions, R2 is updated to point to the character which terminated the conversion. With the exception of the FILNAM call, none of these calls require any arguments. Conversion results are always delivered back to the user in register R1.

9.1 ALF - TEST A CHARACTER FOR ALPHABETIC

The character indexed by R2 is tested for alphabetic (A-Z) and the Z-flag is set if it is or cleared if it is not. R2 is not changed.

9.2 NUM - TEST A CHARACTER FOR NUMERIC

The character indexed by R2 is tested for numeric (0-9) and the Z-flag is set if it is or cleared if it is not. R2 is not changed.

9.3 TRM - TEST A CHARACTER FOR TERMINATOR

The character indexed by R2 is tested for a legal terminator defined as a blank, tab, comma, semicolon, carriage-return, or line-feed. The Z-flag is set if the character is a terminator and cleared if it is not. R2 is not changed.

9.4 LIN - TEST A CHARACTER FOR LINE TERMINATOR

The character indexed by R2 is tested for a legal end-of-line defined as a semicolon, carriage-return, or line-feed. The Z-flag is set if the character is an end-of-line character and cleared if it is not. R2 is not changed.

9.5 BYP - BYPASS BLANKS

Index R2 is advanced past all characters which are blanks or tabs and left indexing the first non-blank, non-tab character it finds.

9.6 GTDEC - INPUT A DECIMAL NUMBER

Index R2 is used to process a decimal number whose value may be from 0-65535 in the input line (leading zeros are legal) and deliver the resultant binary value back in R1. The N-flag will be set if there was an error (result was greater than 65535). R2 will be updated to point to the character following the decimal input number. In the case of an error, R2 will be left indexing the digit that would have caused the overflow past 65535 for double-word processing techniques.

9.7 GTOCT - INPUT AN OCTAL NUMBER

Index R2 is used to process a octal number whose value may be from 0-177777 in the input line (leading zeros are legal) and deliver the resultant binary value back in R1. The N-flag will be set if there was an error (result was

greater than 177777). R2 will be updated to point to the character following the octal input number. If J.HEX is set for this job (via the SET HEX command) this call will process input in hexadecimal instead of octal.

9.8 GTPPN - INPUT A PROJECT-PROGRAMMER NUMBER

Index R2 is used to process a project-programmer number in the standard format of proj,prog and deliver the resultant binary code back in R1. The format dictates that project numbers be octal numbers with a value between 1-377, and programmer numbers be octal numbers with a value between 0-377. The N-flag will be set if the PPN was not in valid format. R2 will be updated to point to the character following the PPN.

9.9 FILNAM - INPUT A FILENAME

Index R2 is used to process a filename.extension input string and leave the RAD50 packed 3-word result in the 3 words starting with the address specified as the first argument of the call. This argument is a standard monitor call argument in format. The second argument is a 1-3 character extension which is to be used in case no explicit extension is entered in the input string. R2 is updated to index the terminating character. The Z-bit will be set if there was no filename to process (the first character was not a legal RAD50 character).

CHAPTER 10

MISCELLANEOUS MONITOR CALLS

This section deals with the monitor calls which don't seem to have a home in any other section covered thus far.

10.1 EXIT - RETURN TO AMOS COMMAND LEVEL

This is the normal means that a program uses to terminate processing and return to monitor command mode. The EXIT call takes no arguments. The monitor, upon executing the EXIT call, will delete all temporary memory modules in the user partition and reset any parameters that are program dependent such as JOBICP, JOBBPT, etc. All assigned devices are also released at this time. The user terminal is then placed in the monitor command mode ready to process another operator command.

10.2 SLEEP - PUT JOB TO SLEEP

This is a simple call which puts the user job to sleep for a specified

number of line clock ticks. The argument is a standard argument which specifies how many clock ticks to sleep for. After the specified number of clock ticks have elapsed the job is automatically awakened and execution proceeds with the instruction following the SLEEP call. Caution - a sleep call with an argument of zero clock ticks will put the job to sleep for about 18 minutes (65536 clock ticks) due to the nature of processing the call. The normal AM-100 system runs with a clock frequency of 60 Hz so each clock tick therefore has a value of 16.7 milliseconds. Also, the first clock tick may occur any time within the first 16.7 milliseconds (not necessarily a full clock tick).

Remember that SLEEP takes a standard argument, therefore to sleep for 1 minute, you would execute a

```
SLEEP   #3600.
```

not a

```
SLEEP   3600.
```

Leaving off the pound-sign (#) is one of the most frequently made coding errors.

10.3 CTRLC - BRANCH ON CONTROL-C

Whenever a control-C is entered on a terminal keyboard (usually to abort a program) no action takes place immediately but rather a flag is set in the JCB status word which must be tested later by the program. The CTRLC call is used within an application program to check the status of the control-C flag (in the JCB status word) and branch to a specific address if the flag is set. This call is a convenience since the user could perform the same task with a few instructions by locating his own JCB status word and checking the J.CCC flag within it. The format of this call is:

```
CTRLC   routine-address
```

where routine-address is the address to branch to within the program if the control-C flag is set.

The CTRLC call does not reset the J.CCC flag but merely indicates that it is set (this allows nested routines to unwind themselves correctly). The user program must then reset the flag explicitly by clearing it in the JCB status word or implicitly by performing the EXIT call which kills the program and returns to monitor mode, clearing J.CCC.

APPENDIX A

DISK STRUCTURE FORMAT

The AMOS monitor supports a flexible disk file system which relieves the programmer of the task of keeping track of files, links and record counts. The structure of the standard disk format used in the AMOS system will be described here for those programmers who wish to do some disk file manipulation or system software programming.

A.1 PHYSICAL RECORD FORMAT

The logical record size for all disks used within the AMOS file structure, regardless of type, is 512 bytes. The hard-disk structures (such as the AM-500 or Trident subsystems), and the AMS floppy format all define the physical record size to be this 512-byte logical record size for efficiency. To maintain compatibility with other systems, the standard IBM- compatible floppy disk format is somewhat different and will be expained in more detail here.

The standard IBM-compatible floppy disk has 2002 128-byte physical records on 77 tracks, each track having 26 sectors numbered 1 through 26. The AMOS system uses a logical record size of 512 bytes (256 words) for each record so the actual record is made up of four standard size 128-byte records on the floppy disk itself. The disk driver routine is responsible for translating the AMOS record number (0-499) to the proper four physical records on the disk. There are only 500 records of 512 bytes each as far as the programmer is concerned and the last two 128-byte records on the floppy disk are lost to his use.

The driver translates the AMOS record number into a starting record number which is four times as great. In addition, a physical sector interleave factor is used so that a 512-byte record requires only one rotation of the disk instead of four which would be the case if an attempt was made to access four physically contiguous sectors on the floppy disk. The interleave factor is 5 meaning that there are four sectors between each logically contiguous pair of sectors.

A.2 DISK RECORD TYPES

There are six different record types in use in the AMOS system, categorized by their use in the logical processing of files. Each record is 512 bytes long but their internal structure differs due to different usage in the system. The six record types are:

1. Disk ID record
2. Bitmap records
3. Master File Directory record (MFD)
4. User directory records
5. Sequential file data records
6. Contiguous file data records

The following three record types take care of records 0-2 which are the same on all disks. Initializing the disk by using the "I" command in the SYSACT program will write out record 1 (empty MFD of all zeros) and record 2 (bitmap with records 0-2 allocated) which logically clears the disk of all users and files and makes all remaining records (3-499) available. These records are then allocated as either user directory records or file data records.

A.2.1 The Disk ID Record

The Disk ID record is always record 0 and is not currently used by the AMOS system. It has been reserved for use by user routines which may want to store disk identification information in it. It is permanently allocated so it will not accidently be used as a data record by any system routine. Alpha Micro will be using this record in the future. If users require the use of this space for a specific application, it is suggested that the user start allocating space at the end of the disk ID record so as to minimize any conflict with Alpha Micro.

A.2.2 The Bitmap

The bitmap is one or more records which always begin with record 2 and extend into as many sequential records as necessary to represent the entire disk. Each word in the bitmap is capable of representing the state of 16 logical records with one bit being used for each record. The bit is set if the record is in use and cleared if it is free. The last two words of every bitmap are a double-word hash total used to maintain bitmap integrity during processing. Any remaining words in the last bitmap record are unused. The bitmap itself is permanently allocated but contains no links to other system disk records. If you destroy the bitmap you can run the DSKANA program to recover it.

A.2.3 The Master File Directory

The master file directory record is always record 1 and forms the root of the file structure tree. It contains one entry of four words for each user PPN which is allocated to this disk by the SYSACT program. A maximum of 63 users may be allocated on any one disk since only one MFD record is available.

A.2.4 The User File Directory

User directory records contain up to 42 entries of six words each to describe user files in the corresponding PPN. The first word of each directory record is a link word to the next directory record in the event that more than 42 files are allocated in the current user area. The final directory record will have a zero link word indicating no more directory records follow.

A.2.5 Sequential File Data Records

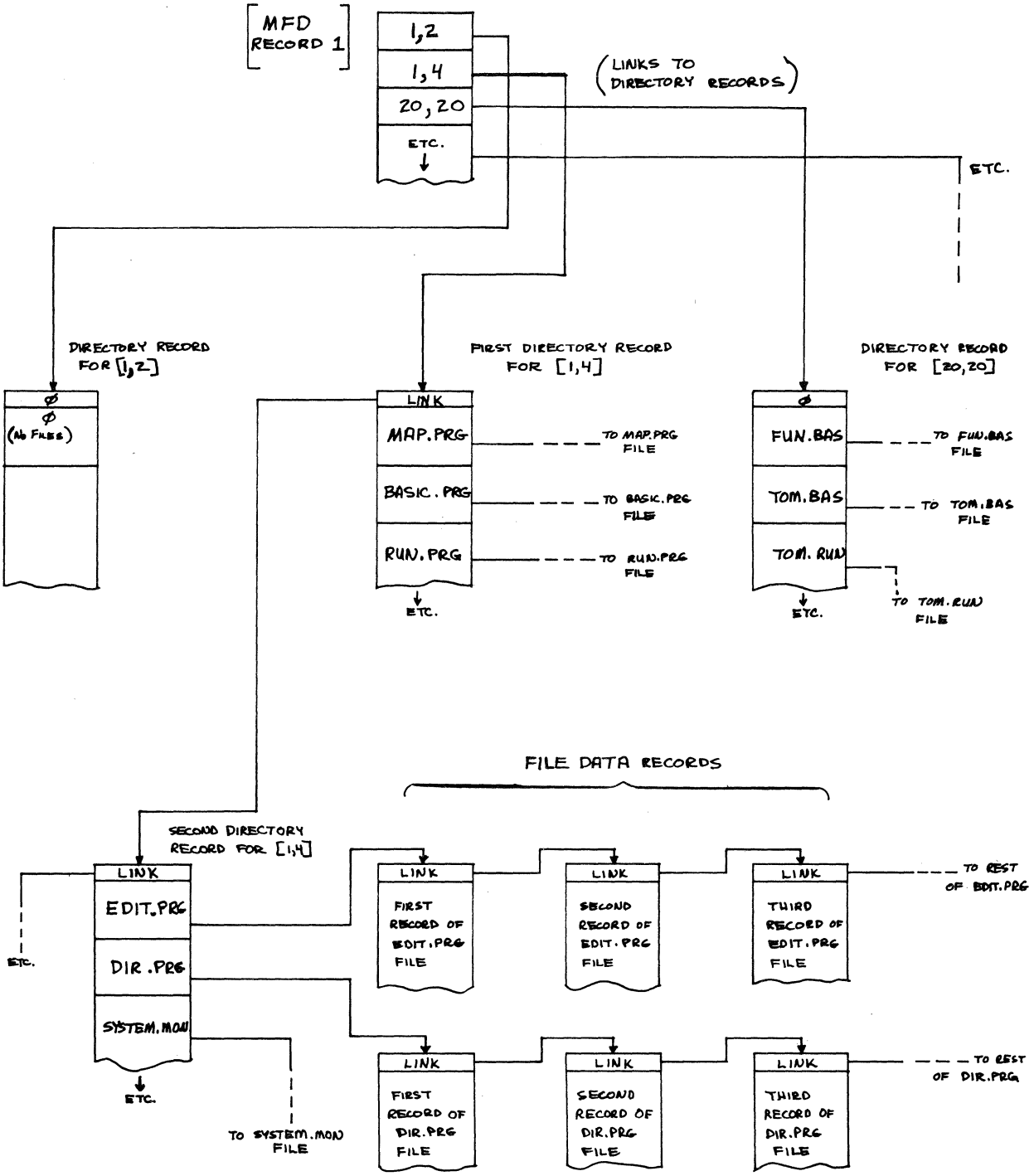
Sequential file data records have a link word and 255 data words. The link word is the record number of the next record in the file. A zero link word indicates this is the last record in the file. The last record in the file may have anywhere from 0-509 active data bytes in its data area. The directory record item contains this number. Sequential files are normally processed as one long string of bytes from start to finish.

A.2.6 Contiguous File Data Records

Contiguous file data records have 256 data words and no links. Contiguous files must be allocated as a block of records with no intervening records belonging to other files. Contiguous files must be preallocated before their use while sequential files are allocated one record at a time as they are required. Contiguous files allow random access processing since any record may be located as a direct offset relative to the base record.

A.3 FILE STRUCTURE

The file struture is depicted in figure A-1 and resembles a tree with the MFD record as its root. The MFD record has one item for each allocated user on this disk. Each MFD item then contains the record number of the first user directory record for that PPN number. The user directory record has one item for each data file in this user's area. Each directory item then contains the record number of the first data record in he file. Sequential



Disk File Structure

Fig A-1

files then chain through the data records by link words as shown in the diagram. The two files that are partially depicted are EDIT.PRG and DIR.PRG in user area [1,4] which just happens to be the system program area. Contiguous files have no link words and must occupy physically adjacent records beginning with the first record as addressed in the directory item. Contiguous files are not depicted in the diagram since they are so straightforward in organization.

A.4 MFD ITEM FORMAT

Each MFD item is four words long and contains the PPN, user directory link, and password. The format of the item is:

Word 1 - user PPN (proj and prog are each one byte)
Word 2 - record number of first user directory record
Words 3-4 - password packed RAD50 (up to 6 characters)

Word 2 is zero if no files have been allocated to this user yet meaning no directory records have yet been allocated. Words 3-4 are zero if no password is required to gain access to this user account when logging on via the LOG command.

MFD items are added, deleted, and changed by the SYSACT program.

A.5 UFD ITEM FORMAT

Each user directory item is six words long and contains information about the data file which it defines. The format of the item is:

Words 1-3 - filename.extension of the file packed RAD50
Word 4 - number of data records in this file
Word 5 - number of active data bytes in last record
Word 6 - record number of first data record in file

Word 1 is -1 (octal 177777) if this file has been erased and the directory item is available for another file definition. Word 1 is zero to mark the logical end of the user directory. The byte count in word 5 is negative if this is a contiguous file. It will also represent the negative active byte count of the file if the contiguous file has been opened for output and written into sequentially.

APPENDIX B

SYSTEM COMMUNICATION AREA

There is an area in monitor memory starting at location 100 (octal) which is called the system communication area. It is defined mnemonically in SYS.MAC and contains specific parameters that deal directly with singular system resources and root addresses. The user should not mess around with these parameters because he might create a disaster. They are briefly defined here for those users who wish to gingerly reference them for whatever diabolical reasons programmers dream up. All references to these parameters should be made symbolically in the absolute addressing mode. An example: the instruction MOV @#JOBTBL,R0 should be used to set the base of the user job table into index register R0.

B.1 SYSTEM - SYSTEM ATTRIBUTES WORD

This word is used to contain system attribute and status flags. Currently it is only used to indicate that the system has been properly loaded when bit 0 is set on. The user should not count on this remaining the case in future releases, however.

B.2 DEVTBL - ADDRESS OF THE DEVICE TABLE

Set up by the DEVTBL program in the system initialization command file this word contains the absolute address of the device table in monitor memory. The format of this table remains a mystery to all.

B.3 DDBCHN - ACTIVE DDB CHAIN

This is the base of the active DDB chain for interrupt driven routines. It is set up and altered by the file service routines as new I/O DDB's are queued for transfer requests and goes to zero each time that there are no requests pending. Not used for non-interrupt driven devices.

B.4 MEMBAS & MEMEND - USER MEMORY POINTERS

These two words define the beginning and end of the complete user memory area. MEMBAS is the address of the first word following the complete resident monitor including the system memory area for user resident programs. MEMEND is the address of the last word in the total physically contiguous RAM memory in the machine. It is set up by the INITIA program when the monitor first starts up by a memory scan technique which locates the last available 1K bank. If memory management is active, MEMEND can only reflect the end of switchable memory within bank 0 and its use in the system diminishes.

B.5 SYSBAS - BASE OF SYSTEM MEMORY

This is the address of the system memory area which is used to contain any user programs set up by the SYSTEM command in the system initialization command file. It is zero if no system memory area exists.

B.6 JOBTBL - ADDRESS OF THE JOB TABLE

This is the address of the user job table which contains one JCB entry for each user allocated via the JOB command in the system initialization command file. For a complete description of the job table and JCB entries refer to the section titled JOB SCHEDULING AND CONTROL SYSTEM.

B.7 JOBCUR - JCB ADDRESS OF THE CURRENT JOB

This word always contains the address of the JCB for the job that is currently running and has control of the CPU. For the user program, it will always point to your own JCB as long as you are running. Obviously if you are referencing this word you must be running. JOBCUR is updated only by the job scheduler in the timesharing monitor.

B.8 JOBESZ - JOB TABLE ENTRY SIZE

This word is set up when the monitor is built and contains the size in bytes of the JCB entry in the job table. This way, when the JCB item expands the programs which scan the job table will not have to be reassembled since they get the JCB size dynamically from JOBESZ. This includes routines within the monitor itself.

B.9 TIME - THE TIME OF DAY

This is a two-word field which is incremented each time the line clock interrupts. It represents the current time of day, which is stored as the number of ticks since midnight. It is the parameter which you can reference if you want to keep track of the time it takes to do something on the machine. Remember, TIME is used to count clock ticks and not seconds or milliseconds. To calculate the actual time in seconds you should divide the elapsed time in ticks by the clock frequency which is stored in the CLKFRQ constant described further on. This of course optimistically assumes that the CLKFRQ command has been used in the system initialization command file to properly set the constant up for your particular frequency (50 Hz overseas, remember?).

B.10 DATE - THE SYSTEM DATE

This is a two-word field which is used by various date routines to store the current date in some specific format. Its use is dependent upon the applications which are defining the format. The DATE field is not accessed or altered by the system monitor itself.

B.11 HLDTIM - THE HEAD LOAD TIMER

This is a two-word area which controls the head-load timing for the AM-200 floppy disk system when used with the Persci Floppy Disk Drive. The second word (at HLDTIM+2) is set up by the HEDLOD program in the system initialization command file to the number of clock ticks desired to wait before unloading the disk heads during periods of inactivity. Each time the head is loaded or another disk transfer is initiated, the count in the second word is transferred to the first word. Each time the clock interrupts, the count in the first word is decremented and if it ever gets to zero the head is unloaded.

B.12 CLKFRQ - LINE CLOCK FREQUENCY

This word is set up by the CLKFRQ command in the system initialization command file to contain the frequency at which the line clock is running. It is used by routines which compute elapsed time based on counting the clock ticks in the TIME constant. Normally set to 60 for systems in North American countries and to 50 for systems running overseas. Achtung!

Remember that CLKFRQ specifes only the local line frequency. Changing CLKFRQ will have no effect on the execution speed of the computer.

B.13 SPXSAV - STACK POINTER SAVE LOCATION

This word is used by the clock interrupt routine for saving the user stack pointer just prior to switching to the internal stack.

B.14 SPXINT - INTERNAL STACK

The address of the internal work stack used for processing clock interrupts. It is set up by the initial load routine and used by the clock interrupt processor.

B.15 LPTQUE - LINE PRINTER SPOOLER QUEUE

The dynamic link address to the base of the line printer spooler queue. The format of the spooler queue is subject to frequent change, so it is not detailed here.

B.16 TRMDFC - BASE OF THE TERMINAL DEFINITION TABLE

The link to the base of the terminal definition table. There is one entry in this table for each terminal defined at system startup by a TRMDEF statement in the SYSTEM.INI file.

B.17 TRMIDC - ADDRESS OF FIRST INTERFACE DRIVER

The link to the first terminal interface driver defined in the system. Each driver then links to the next one in the chain.

B.18 TRMTDC - ADDRESS OF FIRST TERMINAL DRIVER

The link to the first terminal driver defined in the system. Each driver then links to the next one in the chain.

B.19 TRMSCN - THE NON-INTERRUPT TERMINAL QUEUE

The link to the chain of queue blocks for all terminals which are defined as being non-interrupt driven and requiring terminal scan service each clock tick.

B.20 CLKQUE - THE CLOCK QUEUE

The link to the clock queue which will get scanned every clock interrupt. This queue has some entries which will remain constant and some entries which will be continuously added and deleted (such as SLEEP command queue blocks). CLKQUE is actually the base entry in the queue chain and therefore is two words in size.

B.21 SCNQUE - THE IDLE SCAN QUEUE

The link to that point within the clock queue chain which defines the idle scan queue or that portion of the clock queue which will be continuously scanned when the system is idle. SCNQUE is actually the base entry in the queue chain and therefore is two words in size.

B.22 RUNQUE - THE JOB SCHEDULING QUEUE

A 5-word block which forms the base and end entries for the job scheduling and run queue along with the necessary control information. Its format is unimportant to the user, who must never alter it.

B.23 DRVTRK - THE DRIVE/TRACK TABLE

A 4-byte block which is used to store head track positioning information for floppy disks used in the system. It is used only by the head unload and head positioning routines in various floppy disk drivers.

B.24 MEMDEF & MEMBNK - MEMORY MANAGEMENT CONTROL

These two words are used by the memory management system (when active) to store the base of the memory bank definition table and the currently active bank index. They are explained in detail in the section dealing with memory control.

B.25 ZSYDSK - ADDRESS OF SYSTEM DISK DRIVER

This word contains the base address of the system disk driver within the monitor. It is used by MONGEN to overlay the disk driver with another one when changing the resident disk type.

B.26 QFREE - QUEUE SYSTEM CONTROL

QFREE consists of two words, the first containing the number of queue blocks currently available, the second pointing to the first available queue block. Queue blocks are allocated and deallocated by getting and returning them from the front of the list controlled by this address, automatically incrementing or decrementing the free count in the process. The operation of the queue system is more fully explained in Chapter 5.

APPENDIX C

ALPHABETICAL LISTING OF AMOS MONITOR CALLS

The following is a quick reference to all AM-100 monitor calls:

ALF	tests the character indexed by R2 for alphabetic
ASSIGN	assigns a non-sharable device to a job
BNKSWP	changes banks when running under memory management system
BYP	bypasses all spaces and tabs in the string indexed by R2
CHGMEM	changes the size of a user memory module
CLOSE	closes a logical dataset
CRLF	prints a carriage-return line-feed pair on the user terminal
CTRLC	checks for a control-c pending
DCVT	converts a binary value to decimal and prints it on the user terminal
DEASGN	deassigns a non-sharable device from a job
DELETE	deletes a file from a file-structured device
DELMEM	deletes a user memory module from his partition
DSKALC	allocates next available record on disk and returns block number
DSKBMR	reads disk bitmap and sets reentrant lock for user modification
DSKBMW	rewrites disk bitmap after user modification
DSKCTG	allocates a contiguous file for random processing
DSKDEA	deallocates a record on disk and makes it available for use again
DSKDRL	sets reentrant directory lock for a specific user's directory
DSKDRU	clears reentrant directory lock for a specific user's directory
EXIT	exits from user program and returns to monitor command mode
FETCH	fetches a module from disk into user memory unless already in memory
FILNAM	processes a filename specification indexed by R2 into RAD50 format
FSPEC	processes a complete file specification indexed by R2 and sets up DDB
GETMEM	allocates a user memory module in his partition
GTDEC	converts a decimal number indexed by R2 into binary and returns it in R1
GTOCT	converts an octal number indexed by R2 into binary and returns it in R1
GTPPN	converts a PPN format indexed by R2 into binary and returns it in R1
HTIM	sets up the diskette head unload timer function
INIT	initializes a dataset driver block (DDB) for I/O processing
INPUT	performs a logical record input I/O function on an open dataset
JOBGET	retrieves a job control block item for the current job
JOBIDX	set an index to a job control block item for the current job
JOBSET	sets data into a job control block item for the current job

JRUN	restores a waiting job to the run request state
JWAIT	sets an active job into the wait state
KBD	accepts input from user terminal keyboard (character or line mode)
LIN	tests the character indexed by R2 for valid end-of-line character
LOCK	locks the processor against interrupts (perfoms IDS instruction)
LOOKUP	looks for a specific file on disk and returns information about it
NUM	tests the character indexed by R2 for numeric
OCVT	converts a binary value to octal and prints it on the user terminal
OPEN	general form of the I/O logical dataset open calls
OPENI	opens a logical dataset for input
OPENO	opens a logical dataset for output
OPENR	opens a logical dataset for random access
OUTPUT	performs a logical record output I/O function on an open dataset
PACK	packs an ASCII triplet into its RAD50 code
PFILE	prints a complete file specification on user terminal from a DDB
PRNAM	prints a filename specification on user terminal from its packed format
PRPPN	prints a PPN specification on user terminal from its packed format
PTYIN	forces one character into another job's terminal input buffer
PTYOUT	retrieves one character from another job's terminal output buffer
QADD	adds a queue block to the end of a queue list
QGET	gets a queue block from the free list and clears it for use
QINS	inserts a queue block into a queue list at a defined point
QRET	removes a queue block from a queue list and returns it to the free list
READ	performs a physical record read I/O function on a dataset
RENAME	renames a file on a file-structured device
SCAN	forces a single scan of the idle scanner queue (SCNQUE)
SLEEP	puts the user job to sleep for a specified number of line clock ticks
SRCH	searches for a named memory module and returns its address
TAB	sends a tab character to the user terminal
TBUF	queues up a variable length data buffer for output to a terminal
TCRT	executes the special function CRT routine in the active terminal driver
TIN	reads one character from the user terminal input buffer
TOUT	sends one character to the user terminal output buffer
TRM	tests the character indexed by R2 for a valid termination character
TRMBFQ	adds a data buffer to the active output queue of a terminal
TRMICP	processes one input character (used within terminal drivers)
TRMOCP	processes one output character (used within terminal drivers)
TTY	outputs one character to the user terminal
TTYI	outputs an inline message to the user terminal
TTYIN	retrieves one character from any job's terminal input buffer
TTYL	outputs a message to the user terminal
TTYOUT	forces one character into any job's output buffer
TYPE	types an ASCII message on the user terminal
TYPECR	types an ASCII message on the user terminal with appended CRLF pair
TYPESP	types an ASCII message on the user terminal with one appended space
UNLOCK	unlocks the processor for interrupts (performs IEN instruction)
UNPACK	unpacks a RAD50 code word into its equivalent ASCII triplet

USRBAS	returns the address of the current user's memory partition base
USREND	returns the address of the current user's memory partition end
USRFRE	returns the address of the current user's free memory area
WAIT	puts user job into a wait state until the completion of I/O
WRITE	performs a physical record write I/O function on a dataset

Index

ALF 9-1
ASSIGN 6-16

Bitmap Format 6-18
Bitmaps A-2
BNKSWP 3-11
BYP 9-2

CHGMEM 3-6
CLKFRQ B-4
CLKQUE B-5
Clock Frequency B-4
CLOSE 6-11
Contiguous Files A-3
Control-C 10-2
Convenience Macros 7-8
CRLF 7-4
CTRLC 10-2
Cursor Addressing 7-7

DATE B-3
DCVT 8-1
DDB 6-1
  Buffer Address 6-4
  Buffer Index 6-4
  Buffers 6-6
  Call Level 6-6
  Device Code 6-5
  Drive 6-5
  Driver Work Area 6-6
  Error Code 6-2
  Error Handling 6-7
  Extension 6-6
  Filename 6-6
  Flags 6-4
  JCB Address 6-5
  Job Priority 6-5
  Open Code 6-6
  PPN 6-6
  Queue Chain Link 6-5
  Record Number 6-5

Record Size 6-4
DDB Format 6-2
DDBCHN B-2
DEASGN 6-17
Decimal Input 9-2
Decimal Output 8-1
DELETE 6-16
DELMEM 3-6
DEVTBL B-1
DEVTBL program B-1
Disk File Structure A-3
Disk ID Record A-2
Disk Record Types A-2
Disk Service Monitor Calls . . . 6-17
Disk Structure A-1
DSKALC 6-20
DSKBMR 6-20
DSKBMW 6-20
DSKCTG 6-19
DSKDEA 6-20
DSKDRL 6-21
DSKDRU 6-21

EXIT 10-1

FETCH 4-1
Flags 4-2
File Service Monitor Calls . . . 6-8
File Service System 6-1
File Structure A-3
Filenames 8-4, 9-3
Filespecs 8-4
FILNAM 9-3
FORCE command 7-5
FSPEC 6-8

GETMEM 3-6, 6-6
GTDEC 9-2
GTOCT 9-2
GTPPN 9-3

Head Load Time B-3
HEDLOD program B-3
Hexadecimal Input 9-2
Hexadecimal Output 8-1
HLDTIM B-3

INIT 6-6, 6-9
INPUT 6-13
Input Line Processing Calls . . . 9-1
Interface Driver B-4

Interface Drivers . . . . . 7-5 to 7-6

JCB 2-1, B-2
  Size B-3
JCB Entries
  JOBBAS 2-5
  JOBBNK 2-7, 3-11
  JOBBPT 2-7
  JOBBRK 2-8
  JOBCMS 2-6
  JOBCMZ 2-6
  JOBCUR 2-1
  JOBDEV 2-7
  JOBDRV 2-7
  JOBDYS 2-9
  JOBERC 2-6
  JOBFPE 2-8
  JOBNAM 2-4
  JOBPRG 2-5
  JOBPRV 2-5
  JOBRNQ 2-8
  JOBSIZ 2-5
  JOBSPR 2-4
  JOBSTK 2-9
  JOBSTS 2-4
  JOBTRM 2-7
  JOBTYP 2-6
  JOBUSR 2-5
Job Control Block 2-1, B-2
  Size B-3
Job Table B-2
JOBBAS 2-5
JOBBNK 2-7, 3-11
JOBBPT 2-7
JOBBRK 2-8
JOBCMS 2-6
JOBCMZ 2-6
JOBCUR 2-1, B-2
JOBDEV 2-7
JOBDRV 2-7
JOBDYS 2-9
JOBERC 2-6
JOBESZ B-3
JOBFPE 2-8
JOBGET 2-1, 2-3
JOBIDX 2-1, 2-3
JOBNAM 2-4
JOBPRG 2-5
JOBPRV 2-5
JOBRNQ 2-8
JOBSET 2-1, 2-3

JOBSIZ 2-5
JOBSPR 2-4
JOBSTK 2-9
JOBSTS 2-4
JOBTBL B-2
JOBTRM 2-7
JOBTYP 2-6
JOBUSR 2-5
JRUN 2-3
JWAIT 2-3

KBD 7-2, 9-1

LIN 9-2
Line Printer Spooler B-4
LOOKUP 6-10
LPTQUE B-4

Master File Directory A-3, A-5
MEMBAS B-2
MEMBNK B-6
MEMDEF B-6
MEMDEF Program 3-9
MEMEND B-2
Memory Management 3-9, B-6
Memory Mapping 3-9
Memory Modules 3-5, 4-1
Memory Partitions 3-2
MFD A-3, A-5
Miscellaneous Monitor Calls . . . 10-1
Monitor Calls
  ALF 9-1
  Arguments 1-2
  ASSIGN 6-16
  BNKSWP 3-11
  BYP 9-2
  Calling Format 1-1
  CHGMEM 3-6
  CLOSE 6-11
  CRLF 7-4
  CTRLC 10-2
  DCVT 8-1
  DEASGN 6-17
  DELETE 6-16
  DELMEM 3-6
  Disk Service 6-17
  DSKALC 6-20
  DSKBMR 6-20
  DSKBMW 6-20
  DSKCTG 6-19
  DSKDEA 6-20

DSKDRL 6-21
DSKDRU 6-21
EXIT 10-1
FETCH 3-5, 4-1
File Service 6-8
FSPEC 6-8
GETMEM 3-6, 6-6
GTDEC 9-2
GTOCT 9-2
GTPPN 9-3
INIT 6-6, 6-9
INPUT 6-13
Input Line Processing 9-1
JOBGET 2-1, 2-3
JOBIDX 2-1, 2-3
JOBSET 2-1, 2-3
JRUN 2-3
JWAIT 2-3
KBD 7-2, 9-1
LIN 9-2
LOOKUP 6-10
Memory Control 3-1
Miscellaneous 10-1
NUM 9-2
Numeric Conversion 8-1
OCVT 8-1
OPENI 6-10
OPENO 6-11
OPENR 6-11
OUTPUT 6-14
PACK 8-4
PFILE 8-4
Printing Conversion 8-4
PRNAM 8-4
PRPPN 8-5
PTYIN 7-5
PTYOUT 7-5
QADD 5-3
QGET 5-3
QINS 5-3
QRET 5-3
RAD50 Conversion 8-3
READ 6-11
RENAME 6-16
SLEEP 10-1
SRCH 3-5, 4-1
Standard Address Argument . . . 1-2
TAB 7-4
TBUF 7-6
TCRT 7-7
Terminal Service 7-1

TIDX (obsolete) 2-8
TIN 7-3
TOUT 7-3
TRM 9-2
TRMBFQ 7-6
TRMICP 7-5
TRMOCP 7-6
TTY 7-3
TTYI 7-4, 7-8
TTYIN 7-5
TTYL 7-4
TTYOUT 7-5
TYPE 1-1
UNPACK 8-4
USRBAS 3-2
USREND 3-2
USRFRE 3-2
WAIT 6-15
WRITE 6-12

NUM 9-2
Numeric Conversion Monitor Calls 8-1
Numeric Input 9-2

Octal Input 9-2
Octal Output 8-1
OCVT 8-1
OPENI 6-10
OPENO 6-11
OPENR 6-11
OUTPUT 6-14

PACK 8-4
PFILE 8-4
Physical Disk Record Format . . . A-1
PPNs 8-5, 9-3
Printing Conversion Monitor Calls 8-4
PRNAM 8-4
Project-Programmer Numbers . . . 8-5, 9-3
PRPPN 8-5
Pseudo Terminals 7-5
PTYIN 7-5
PTYOUT 7-5

QADD 5-3
QFREE B-6
QGET 5-3
QINS 5-3
QRET 5-3
QUEUE command 5-2
Queue System 5-1, B-6

Manipulating Queue Blocks . . . 5-3
Obtaining a Free Queue Block . 5-3
Returning a Queue Block 5-3

RAD50 Conversion Monitor Calls . 8-3
Random File Processing 6-11, 6-14
Random Files A-3
READ 6-11
RENAME 6-16
RUNQUE B-5

SCNQUE B-5
Sequential Files A-3
SLEEP 10-1
SPXINT B-4
SPXSAV B-4
SRCH 3-5, 4-1
Flags 4-2
Standard Address Argument 1-2
SYS.MAC 1-1, 2-1, 7-8
SYSBAS B-2
SYSTEM B-1
System Communication
QFREE 5-1
System Communication Area B-1
CLKFRQ B-4
CLKQUE B-5
DATE B-3
DDBCHN B-2
DEVTBL B-1
DRVTRK B-5
HLDTIM B-3
JOBCUR B-2
JOBESZ B-3
JOBTBL B-2
LPTQUE B-4
MEMBAS B-2
MEMBNK 3-10, B-6
MEMDEF 3-10, B-6
MEMEND B-2
QFREE B-6
RUNQUE B-5
SCNQUE B-5
SPXINT B-4
SPXSAV B-4
SYSBAS B-2
SYSTEM B-1
TIME B-3
TRMDFC B-4
TRMIDC B-4
TRMSCN B-5

TRMTDC B-5
ZSYDSK B-6
System Date B-3

TAB 7-4
TBUF 7-6
TCRT 7-7
Terminal Definition Table B-4
Terminal Driver B-5
Terminal Drivers 7-6
Terminal Input 7-2
Terminal Service Monitor Calls . 7-1
Terminal Status Word 7-2
TIDX (obsolete) 2-8
TIME B-3
Time of Day B-3
TIN 7-3
TOUT 7-3
TRM 9-2
TRMBFQ 7-6
TRMDFC B-4
TRMICP 7-5
TRMIDC B-4
TRMOCP 7-6
TRMSCN B-5
TRMTDC B-5
TTY 7-3
TTYI 7-4, 7-8
TTYIN 7-5
TTYL 7-4
TTYOUT 7-5
TYPE 1-1, 7-8
TYPECR 7-8
TYPESP 7-8

UFD A-3, A-5
UNPACK 8-4
User File Directory A-3, A-5
USRBAS 3-2
USREND 3-2
USRFRE 3-2

WAIT 6-15
WRITE 6-12

ZSYDSK B-6

Alpha Microsystems
17881 Sky Park North
Irvine, California 92714