ALPHA MICROSYSTEMS
RIGHT. FROM THE START.

AMOS®
Monitor Calls

DSM-00040-10

REVISIONS INCORPORATED	
REVISION	DATE
00	March 1988
01	December 1988
02	September 1989
03	April 1991
05	May 1997
06	December 1997
07	May 1998
08	June 1998
09	October 1999
10	April 2000

AMOS Monitor Calls
To re-order this document, request part number DSO-00040-00.

This document applies to AMOS versions 2.3A, PR 10/99, AlphaTCP 1.5A and later

ALPHA MICROSYSTEMS
2722 S. Fairview St.
Santa Ana, CA 92799

Summarized Table of Contents

Table of Contents

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 repetitively write the same routine. Perhaps even more important is the level of compatibility to be gained by having all AMOS software use the same central collection of routines. By having all software perform the same operation the same way, a high degree of compatibility is built into all software with virtually no extra effort on the part of the programmer. This manual describes the monitor calls provided by AMOS.

We assume that the reader of this manual is familiar with assembly language programming and the M68000 family instruction set as it is described in the *Alpha Micro Instruction Set Manual*. We also assume that the reader is familiar with the AMOS macro assembly system described in the *AMOS Assembly Language Programmer's Manual*.

This reference manual is most emphatically **not** a tutorial on assembly language programming. Many such tutorials exist; if you are just learning assembly language, you should consult such a book before reading this manual.

Chapter 1
Communicating with AMOS

One of the primary functions of any operating system is to provide services to the processes running on the system. The AMOS monitor contains hundreds of such service routines available for use by assembly language programs.

Your assembly language programs make use of these routines by using monitor calls. Other languages provide access to these monitor calls through subroutines and functions which themselves make use of the monitor calls. AMOS monitor calls are defined in the form of macros so they are easy to incorporate into your assembly language programs. The monitor call macros are in the system library file SYS.M68 (and its related universal file SYS.UNV) in account [7,7] of the system disk.

The following chapters group monitor calls by the functions they perform. For example, Chapter 8 discusses all monitor calls that do data conversion. For an alphabetical list of all monitor calls see Appendix E.

COMPATIBILITY ISSUES

One of the major benefits of encapsulating commonly performed functions within monitor calls—other than simply saving programming time—is it hides many of the details of the system from you. By doing so, it helps make sure future changes and enhancements made within AMOS do not adversely affect your programs.

This can be very important when you wish to upgrade to the latest version of AMOS to take advantage of the latest whiz-bang feature. If your program is sensitive to every little change, chances are good it won't work after changing versions of AMOS. But, because AMOS hides most of its internal workings within monitor calls, and assuming your program follows the rules, major changes internal to AMOS have no affect on your programs.

So what do we mean by "follows the rules?" Well, we mean it uses the monitor calls outlined within this manual, exactly as defined here, and it follows all the other rules, guidelines, and hints contained in the other several thousand pages of documentation available describing AMOS.

Obviously it is a rather ambitious undertaking to read all that information, commit it to memory, and never deviate from it. Of course there will be times when you didn't read something, forgot it along the way, or simply choose to ignore a rule. As long as your software works, who cares? If you are programming for a single customer—such as yourself—don't ever plan on changing versions of AMOS, don't plan to add software from other sources, and don't ever change your hardware configuration, there is absolutely nothing wrong. If you do want to do some of that, and the rule you didn't read/forgot/ignored turns out not to be important in your case, you are still OK. But somewhere down the line, when you can least afford it, that minor little violation if going to turn around and bite you.

Still, you say, how can you possibly learn every rule and never make a single mistake? Simple, you can't. Virtually every piece of software will end up breaking some rule or other, and in most cases it makes no difference. No team of "code inspectors" is going to arrive on your doorstep to scrutinize every line of code, writing you up for any violations found. It is simply in your best economic interest to try to follow the rules the best you can. Time spent on the rules during the development phase will more than be returned in time saved during debugging, installation, and customer support.

Fortunately, the rules are pretty static. Even as AMOS has moved from processor to processor, the rules have stayed the same. What has changed is what you can get away with. Rule violations that have not been a problem for years may suddenly rear their heads when a new version of AMOS relies on the rule being followed. Alpha Micro goes to great lengths to make sure each new release of AMOS is upwardly compatible. Any software that follows the rules and runs under a prior release should run under the latest release without change. Despite our efforts this may not always be the case (we make mistakes too), but what it does mean is that if your software does not follow the rules, Alpha Micro will not worry about whether it might get broken or not.

At the assembly language level, there are several rules that are particularly important. These are not the only rules, of course, but they are the ones that, when broken, cause the greatest levels of incompatibility. Please read these and treat them as words to live—and program—by!

- **Don't rely on timing loops for fixed delays.** In these days of different relative processor speeds within otherwise compatible systems, using timing loops to generate delays simply will not work. Some processors load small loops into internal cache memory, making timing loops execute very quickly. Use the TIMER or SLEEP monitor call to generate fixed delays.
- **Never access hardware directly.** Unless you are writing a device driver, your software should never access Input/Output addresses. Remember AMOS is a multi-user system so device accesses must be coordinated through the AMOS IO system. Do all IO through standard device drivers, writing one yourself if you need to.
- **Don't use any field referred to in this manual as "reserved," "spare," or "undefined."** All of these words mean the same thing: we may not use it now, but we will as soon as you start to rely on it. In many cases internal data structures are defined with extra space reserved for forthcoming features. If your software makes use of these fields it will fail as soon as AMOS starts to use the same locations. Load a small module into system memory and locate it with a SRCH call if you need a centralized, shared memory resource, or use the monitor calls described in Chapter 4.
- **Don't rely on stack frame sizes.** Particularly within interrupt routines you must be aware of the differences between the members of the 68000 family. Each of these processors uses a slightly different stack frame format. Failure to allow for this will cause your software to fail when moved to different processor types.
- **Don't use self-modifying code.** Some members of the 68000 family perform instruction caching but not data caching. This means any self-modifying code, including the loading of program code into memory, will not properly update the cache, causing your program to execute the wrong instructions.

Remember, it is up to you to decide to follow the rules or not. You may in fact come up with a good reason for deciding not to, but remember one of the main reasons people choose to use an AMOS system is because of the high degree of compatibility shown by different software packages on a system, and by all of those packages over time. If your package requires constant updating and changing, just to work with the latest version of everyone else's software, your software may not be viewed favorably.

MONITOR CALL CALLING FORMAT

Now let's move on to how to use monitor calls. The general format used with all monitor calls is:

```
{Label:} Opcode {Arguments}  {; Comments}
```

As the format shows, the only required item in all of the calls is ***opcode***, which is the name of the monitor call. You may optionally specify ***label***, in which case the assembler assigns the label the address of the first instruction of the monitor call sequence.

Some calls generate several words of code to perform their function. The total number of words a monitor call generates depends on the call itself as well as the addressing modes of any arguments you specify. Those calls which incorporate an ASCII message (e.g., the TYPE call) generate a string of bytes whose length depends on the message involved. As in machine instructions, you may also place comments at the end of the line. Each line of comments must begin with a semi-colon.

Arguments

Some calls require one or more arguments to specify parameters for the execution of the monitor call function and where to return the results of the call. These arguments are of three types:

- Address Pointers (***adr***) that point to the data item on which to operate
- Source Operands (***src***) that supply data to the monitor call directly
- Destination Operands (***dst***) that specify where to place data after the monitor call is finished.

Address Pointers

An address pointer, as used in a monitor call, is an expression that evaluates to a pointer to the data item you are supplying to the monitor call. You can use any valid control effective address to specify an address pointer, including:

`@An`	Register indirect
`x(An)`	Register indirect with offset
`x(An)[Rn]`	Indexed register indirect with offset
`x`	PC-relative
`x[Rn]`	PC-relative with index and offset
`x`	Absolute (word or longword)

The monitor call format descriptions show address pointers as ***adr***.

Typically, you will use an address pointer to specify the location of a DDB, a queue block, or an ASCII string. You will find more information on these data structures in the chapters that follow.

Note that, while the monitor call may modify the data structure being pointed to, it will never modify the address pointer itself.

Source Operands

A source operand, as used in monitor calls, specifies the data item that contains the argument you are passing to the monitor call. You may use any of the addressing modes to specify a source operand. The monitor call format descriptions show a source operand as ***src***.

For some calls, one argument serves as both source and destination operands; in this case, the source operand is restricted to those addressing modes which are valid when specifying destination operands. The monitor call format descriptions show this case as ***src/dst***.

Destination Operands

A destination operand, as used in monitor calls, specifies the data item in which the call will return data. You may use any of the alterable addressing modes to specify destination operands, including:

`Dn`	Data register direct
`An`	Address register direct
`@An`	Register indirect
`(An)+`	Post-increment register indirect
`-(An)`	Pre-decrement register indirect
`x(An)`	Register indirect with offset
`x(An)[Rn]`	Indexed register indirect with offset
`x`	Absolute (word or longword)

The monitor call format descriptions show destination operands as ***dst***.

As before, one argument may serve as both source and destination operands for some calls; in this case, the source operand is restricted to those addressing modes which are valid when specifying destination operands. This case is shown as ***src/dst*** in the monitor call format descriptions.

USE OF MONITOR CALL REGISTERS

The assembler uses the temporary registers as part of the operand processing of the monitor calls. For this reason, we do not guarantee the contents of the temporary registers A6, D6, and D7 will be preserved after monitor call use; in fact, these registers are rarely left unmodified.

MONITOR CALL SYMBOLS (SPECIAL .UNV FILES)

In addition to the monitor call definitions and symbols defined in SYS.M68 (and its related universal file SYS.UNV), there are two additional files that contain definitions of symbols and macros useful to the assembly language programmer. These two files are SYSSYM.UNV and TRM.UNV, both on DSK0:[7,7].

SYSSYM contains definitions for many of the data structures AMOS uses. TRM contains definitions of symbols for the terminal control data structures within AMOS. Because the symbols SYS and SYSSYM contain are used in almost every program, every source program typically uses the SEARCH pseudo opcode to access them.

The examples in this book assume you have used the SEARCH pseudo opcode within your source program to access SYS.UNV, SYSSYM.UNV, and TRM.UNV.

Chapter 2 Job Scheduling and Control System

The AMOS timesharing operating system allocates jobs and schedules CPU time and resources for them based on their processing requirements. In order to write assembly language programs which make use of some of the more complex features of the system, you should have a basic understanding of how AMOS schedules and controls jobs. The theory behind job-handling is more complex than we can cover in one section of this manual, but we can explain the fundamentals of job control by user programs.

Each job running in the system has three dedicated components which are not shared by any other job in the system: a job table entry, a job control block, and a user memory partition. The job table in the monitor memory area contains one entry for each job *allocated* on the system. Each entry in this job table is a longword, and points to a job control block (JCB) created for each job *assigned* on the system.

In the system initialization command file, the JOBS command allocates the number of jobs on the system, and the JOBALC command assigns names to them. The entry for each job contains specific information about that job. Note it is possible for you to allocate more jobs on the system than you assign.

For example, you may want to allocate 50 jobs in the JOBS command but only assign job names to 30 of them in the JOBALC command. Thus, the job table may contain empty entries which do not point to a JCB.

THE JOB SCHEDULER

At the heart of AMOS is the system job scheduler. Its task is to allocate time to the various jobs running on the system. To do this, it makes use of two different real-time clocks: one to schedule jobs (the *scheduler clock*) and the other to act as a system timer (the *timer clock*).

The scheduler clock runs at an effective rate of 200 kHz, giving a clock period of five microseconds. Each clock period is called a *jtick*. These jticks are grouped into *jiffies* of 257 jticks or 1.285 milliseconds. These jiffies are the basis of job scheduling.

At the standard job priority of 13, the system job scheduler allows each job to run for a maximum of 13 jiffies or 16.7 milliseconds. Each group of 13 jiffies is known as a *quanta*. The number of jiffies per quanta is selectable on a job-by-job basis as the *job priority*. Thus, we have the following values:

1 jtick	= 5 µs
1 jiffy	= 257 jticks = 1.285 ms
1 quanta (default)	= 13 jiffies = 16.7 ms
1 second	= 778.21 jiffies
1 minute	= 46,692.6 jiffies
1 hour	= 2,801,556.4 jiffies
1 day	= 67,237,353.6 jiffies
1 longword	= 63.9 days worth of jiffies

The timer clock runs at a frequency of 10 kHz, giving a clock period of 100 microseconds. Each clock period is called a *tick*.

AMOS's dynamic job priority scheduling feature can adjust the priority for each job as it is run, based on the number of jobs in the run queue. A table contains the priority to give each job based on the current size of the run queue. Thus, a job will get a higher priority (more CPU time) when there are fewer jobs waiting to run.

You can turn dynamic scheduling on and off using the SET DYN option. You can adjust the dynamic job priority table using ADJIT.LIT. See the sheets on these commands in the *AMOS System Commands Reference Manual*.

THE JOB CONTROL BLOCK (JCB)

The format of the JCB is a series of equate statements in the system library file SYS.M68 on DSK0:[7,7]. Each equate statement has the name JOB*xxx*, where ***xxx*** is a 3-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. You may, during the course of your program, wish to read the current data in your own JCB or, in some instances, modify it. You should make references to the JCB items by indexing your JCB by using the JOBIDX monitor call, and then referencing all JCB items by using JOB*xxx*(An).

Three entries in the system communication area define the job control system during timesharing operation. These three entries are not part of the JCB areas, but rather are non-sharable parameters set up during system initialization; they are not part of any one job. We point this out because the names of these three words are JOBTBL, JOBCUR, and JOBESZ. Their names would indicate they are part of a user JCB but they are not.

JOBTBL is a longword that contains the base of the job table, which contains a list of the allocated JCBs. This address is set up at system initialization time and is never changed. The job table itself is an array, each element being a longword. The possible values of each element are:

Value	Meaning
0	There is no JCB corresponding to this entry
-1	Marks the end of the job table
Other values	Address of a JCB

JOBCUR, a longword, always contains the address of the JCB which has 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 always points to your JCB if you reference it, because the reference is only executed while you have control of the CPU. Note, however, you should always locate your own JCB through use of the

JOBIDX monitor call rather than by referencing JOBCUR. Future versions of AMOS will not allow direct reference to JOBCUR.

JOBESZ, a word, contains the size of the JCB in bytes and is used by the system during system initialization.

Example - Scanning the Job Control Area

There are times (such as in a system status report) when you want to scan down the job table and process each JCB. The following example illustrates this process:

```
        MOV     JOBTBL,A0     ; index job table with A0
                              ; loop for each entry in job table:
LOOP1:  MOV     (A0)+,D7      ; get pointer to JCB
        BEQ     LOOP1         ; no job allocated here, get another
        CMP     D7,#-1        ; end of job table?
        BEQ     ENDTBL        ;   yes -
        MOV     D7,A1         ;   no - put index to JCB in A1
                              ; come here to process each JCB
                              ; references to JCB items are by JOBxxx (A1)
LOOP:   ...     ...           ; process JCB indexed by A1
        ...     ...
        ...     ...
        BR      LOOP1         ; go try another
                              ; at this point we have finished
                              ;   the job table scan
ENDTBL: ...
        ...     ...
```

ACCESSING YOUR JCB

You can gain access to your own JCB through use of the JOBIDX monitor call. JOBIDX returns a pointer to the base of your job's JCB in the specified destination. Further access to your JCB should be done by using this base address as an index. The calling format is:

```
        JOBIDX  dst           ; sets absolute address of JCB into dst
```

A typical use of JOBIDX, loading your job's terminal definition pointer into A5, would look like this:

```
        JOBIDX  A6            ; get JCB address into A6
        MOV     JOBTRM(A6),A5 ; get terminal definition pointer
```

ACCESSING ANOTHER JOB'S JCB

When you need to gain access to another job's JCB, you can use the JCBIDX monitor call to locate that JCB. You specify the job whose JCB you wish to index by supplying JCBIDX with an ASCII string giving the job name. The JCBIDX call returns the base address of the JCB in the destination argument you provide. Further access to the JCB should be done by using this base address as an index.

If the specified job is located successfully, the Z-bit will be set upon completion of the call. If no job with the specified name can be located, the call returns with the Z-bit reset. Note this assumes the use of an address register as the destination argument. Use of memory as the destination will corrupt the return flags. The calling format is:

```
              JCBIDX   string,dst
              BNE      job-not-found
```

A typical use of JCBIDX, getting the amount of CPU time used by a job into D1, would look like this:

```
              JCBIDX  NAME,A6         ; get JCB address into A6
              BNE     NOTFND          ; error - job not found
              MOV     JOBCPU(A6),D1   ; get amount of CPU time used
              ...
              ...
      NAME:   ASCIZ   /PRTSPL/        ; name of job to locate
              EVEN
```

JOB SCHEDULING CALLS

Various routines within AMOS use two calls, JWAIT and JRUN, for controlling the job scheduling processes. JWAIT sets your job into the wait state, and JRUN re-activates it to the run state. If your program specifies the J.NXT flag, AMOS places your job at the beginning of the run queue; when your program does not specify J.NXT along with other JRUN flags, AMOS places your job at the end of the run queue.

JRUN requires the job being controlled be indexed by A0 (which must point to the base of the JCB for that job), and the argument specify one of the status control bits (in JOBSTS) to be used as the control flag.

The section on JOBSTS, below, contains the flags used in these calls. The call formats for JWAIT and JRUN are:

```
              JWAIT    flags
              JRUN     flags
```

SLEEP - Put Job to Sleep

SLEEP is a simple call that puts a user's job to sleep for the number of clock ticks you specify in the argument. After the specified amount of time has elapsed, the job automatically awakens and execution continues with the instruction following the SLEEP call. AMOS sets the Z-flag if the job slept for the specified number of clock ticks; AMOS resets the Z-flag if the job woke up prematurely because another job used the WAKE call. The calling sequence is:

```
              SLEEP    src           ; put job to sleep
              BNE      wokeup        ; branch if we were aWAKEned
```

A sleep call with an argument of zero clock ticks puts the job to sleep for approximately 4.97 days (4,294,967,295 clock ticks).

The central processor runs with a timer clock frequency of 10 kHz; each clock tick, therefore, has a value of 100 microseconds. Thus, to sleep for one second, you would specify a value of 10000_{10} to the SLEEP call.

Remember SLEEP takes a standard source argument. Therefore, to cause the job to sleep for one minute, you would execute:

```
              SLEEP    #600000.
```

not:

```
              SLEEP    600000.
```

Leaving off the pound sign (#) is a frequent coding error.

WAKE - Wake Up Job

This call wakes the specified job from a SLEEP call. A0 must point to the base of the JCB of the job you want to wake out of the sleep state. AMOS sets the Z-flag if the call is successful. If the specified job was already awake, AMOS resets the Z-flag. The call format is:

```
              MOV      jcb address,A0   ; index the job to awaken
              WAKE                      ; wake job
              BNE      already awake ; branch if already awake
```

JOB CONTROL BLOCK FORMAT

The following paragraphs describe the entries contained in your JCB. Your programs may access each of these entries by first setting an index to your JCB using the JOBIDX call, and then indexing into the JCB with the symbols shown below.

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 condition that applies to the job. Some legitimate conditions are indicated 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 status of the job changes, so do not alter this word without exercising extreme caution. The following list describes briefly what each bit indicates when it is set.

Symbol	Octal Value	Hex Value	Meaning
J.TIW	2	2	Job is in terminal input wait state
J.TOW	4	4	Job is in terminal output wait state
J.SLP	10	8	Job is in sleep state
J.IOW	20	10	Job is in I/O wait state
J.EXW	40	20	Job is in external event wait state
J.SMW	100	40	Job is waiting on a semaphore
J.CCC	200	80	A Control-C abort is waiting to be processed
J.MSG	400	100	Job is waiting for a message
J.MON	1000	200	Job is in AMOS command mode (no program active)
J.SIW	2000	400	Job is in software interrupt wait state
J.PLK	4000	800	Job is locked out by another job's PLOCK
J.SUS	10000	1000	Job is in suspended state
J.LOK	20000	2000	Job has CPU locked (via the JLOCK monitor call)
J.FIL	40000	4000	Job is waiting for a file or record lock

If any flag other than J.CCC or J.MON is set, the job will not be scheduled for CPU time until those flags have been cleared.

JOBTYP - The Job Type

JOBTYP, one word, specifies the type of processing assigned to the current job. The list below indicates what these flags mean when they are set:

Symbol	Octal Value	Hex Value	Meaning
J.USR	1	1	Job is a user partition
J.NUL	2	2	Job is currently running the null subroutine
J.NEW	4	4	Job is processing a new memory allocation
J.LPT	10	8	Job is running the line-printer spooler (LPTSPL)
J.HEX	20	10	Binary inputs and outputs are in hex (not octal)
J.DER	40	20	Print disk error retry messages
J.VER	100	40	Activate auto-verify mode for disk writes
J.CCA	200	80	Control-C interrupts are enabled (via SET CTRLC)
J.GRD	400	100	Terminal is guarded against SEND and FORCE commands
J.TSK	1000	200	Job is running as a slave task
J.CAB	2000	400	Control-C abort is enabled (reserved for future use)
J.PRE	4000	800	Job was preempted last time it was scheduled
J.PRO	10000	1000	Job preempted another job last time it was scheduled
J.SRQ	20000	2000	Job should save remaining quanta next time it is descheduled
J.PRM	40000	4000	Job is a permanent subtask (do not delete job when parent job exits)
J.NLK	100000	8000	Job has file locking turned off

JOBTY2 - More Job Type Flags

JOBTY2, one longword, specifies the type of processing assigned to the current job. The list below indicates what these flags mean when they are set:

Symbol	Octal Value	Hex Value	Meaning
J2$REM	1	1	Job is a remote connection from another system
J2$AGT	2	2	Job is acting as an agent for a remote task
J2$BTG	4	4	Job is boot job, and system is still booting. (Reset when MEMORY 0 in system initialization file is processed.)
J2$SAP	100	40	Suppress AMOS prompt
J2$TST	400	100	Job is in "test" mode

JOBSPR - The Stack Pointer Reset Address

One longword, JOBSPR, is used to store the user stack pointer reset address which is calculated when the system is initialized. This address is then used to reset the user stack pointer each time the job exits back to AMOS command mode. If you must reset SP yourself, such as within an error recovery routine, you should simply move the contents of JOBSPR to SP. Never modify JOBSPR itself.

JOBNAM - The Job Name

JOBNAM, one longword, contain the 6-character job name packed RAD50. This name is set up by the JOBALC command in the system initialization command file. If a user program alters this word, it effectively alters the name of the job.

JOBBAS - The Memory Base Address

JOBBAS, one longword, 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 de-allocates user memory partitions. We advise against altering this address unless you thoroughly understand the memory allocation process.

JOBSIZ - The Memory Partition Size

JOBSIZ, one longword, 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 defines the current user memory partition. JOBSIZ is altered only by the MEMORY program and the AMOS command processor.

JOBUSR - The Current PPN

JOBUSR, one word, contains the current PPN (account number) if the user is logged in. Zero indicates the job is currently logged off. 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.

JOBPRV - The Privilege Word

JOBPRV, one word, is used to store the privileges associated with the job. This word is set by the LOG and LOGON programs and may be modified, providing the job has the privilege of doing so, by the SET program.

Each bit in this word corresponds to a particular privilege granted to the job. These bits may be checked by individual programs before proceeding with a privileged operation. In addition, AMOS matches these privilege bits against the privilege bits in the program header of any program the job tries to run. If the program requires privileges the job does not currently have, the program will not be run and an error message will be displayed.

The privilege bits are defined as follows:

Symbol	Octal Value	Hex Value	Meaning
PV$RSM	1	1	Job is allowed to read system memory
PV$WSM	2	2	Job is allowed to write system memory
PV$RPD	4	4	Job is allowed to read physical disk blocks
PV$WPD	10	8	Job is allowed to write physical disk blocks
PV$DIA	20	10	Job is allowed to run diagnostic programs
PV$PRV	100000	8000	Job is allowed to change its own privileges

JOBEXI - Job Exit-Trap Stack Pointer

This field contains the value of the stack pointer the last time you executed an AMOS or PCALL monitor call. It is reset to the previous value each time you go through the exit-trap system.

JOBPRG - The Current Program Name

JOBPRG, one longword, contains the 6-character name, packed RAD50, of the program which is currently running, or which was the last job run if you're in AMOS command mode. JOBPRG is loaded by the command processor when the program is loaded or located for execution.

JOBCMZ - The Command File Size

JOBCMZ is one word containing 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. AMOS sets this word to the initial size of a command file when that file is loaded into the top of the user partition and decreases it as each line is extracted from the area and sent to the AMOS 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. Do not alter this word.

JOBCMS - The Command File Status

JOBCMS is one word containing flags used by the command file processor in the low byte and the last character seen by the command file processor in the upper byte. The flags are defined as follows:

Symbol	Octal Value	Hex Value	Meaning
C.MON	1	1	Command file is at AMOS command level
C.SIL	2	2	Command file is in silence (:S) mode
C.TRC	4	4	Command file is in trace (:T) mode
C.KIN	10	8	Command file is in keyboard input (:K) mode
C.PTL	20	10	Command file is in partial input (:P) mode

JOBCMS works in conjunction with JOBCMZ during command file processing.

JOBERC - The Error Control Address

JOBERC, one longword, controls the handling processor exceptions (address errors, privilege violations, bus errors, etc.) as described in the *AM-100/L Instruction Set Manual*. If JOBERC is zero, a processor exception causes a message appropriate to the error to be printed on the user's terminal, and aborts the job. If JOBERC is non-zero, the program jumps to the address specified in JOBERC, which should contain a valid routine for shutting down the program. Note the processor exceptions discontinue the user program only and do not abort the whole timesharing system.

AMOS provides a method for intercepting and handling these processor exception errors. To use this feature, your program must place the address of an error handling routine in the JOBERC longword within the controlling job's Job Control Block. When error handling is not in effect (the default condition), JOBERC must be set to zero.

When a system error occurs, control will be transferred to the error handling routine. The processor will be in supervisor mode. The word at the top of the supervisor stack indicates the type of error which occurred. The remainder of the stack contents varies according to the type of error which occurred, as described below. All registers are exactly as they were at the time the error occurred.

The error handling routine must determine what action to take based on the type of error that occurred. Note many of the most common errors (address error, parity error) cannot be re-started on 68000 processors, although they may be restartable on 68010 and later processors. That is, the processor does not save enough information for your error handling routine to correct the error condition and resume execution. In most cases, it is expected the error handling routine will simply perform clean-up operations appropriate to the application, then perform an EXIT to return control to AMOS.

If after entering your error routine you decide not to handle the error, but wish the system to take its normal exception processing, you can issue the STDERR monitor call to cause the system to resume default error processing. Since the default error processing results in an EXIT call being performed, there is no return from the STDERR call. More detail on the STDERR call may be found in Chapter 13.

If your application requires you to try to resume execution, your routine is responsible for properly saving and restoring all registers, re-setting the stack contents as necessary, and any other required operations. Your routine may then resume execution by using a RTE instruction. You must be extremely careful, as errors within errors typically cause the processor itself to halt, requiring a hardware reset to reboot the system.

The general format of the stack upon entering the error handling routine is:

@SP	Error trap type
2(SP)	CPU status register
4(SP)	Program counter
6(SP)	
10(SP)	Other information depending onexception type

The error trap types are defined as follows:

Octal Value	Hex Value	Symbol	Note	Meaning
1	1	EX$BER	*	Bus error
2	2	EX$PAR	*	Memory parity error
3	3	EX$ADR	*	Address error
4		EX$IIN		Illegal instruction
5	5	EX$DIV		Divide by zero
6	6	EX$CHK		CHK instruction trap
7	7	EX$TRP		TRAPV instruction trap
10	8	EX$PRV		Privilege violation
11	9	EX$TRC		Trace trap (only if JOBTRC is zero)
12	A	EX$EM1		EM1111 instruction trap
13	B	EX$MSC		Miscellaneous exception
14	C	EX$II0		Illegal interrupt on level 0
15	D	EX$II1		Illegal interrupt on level 1
16	E	EX$II2		Illegal interrupt on level 2
17	F	EX$II3		Illegal interrupt on level 3
20	10	EX$II4		Illegal interrupt on level 4
21	11	EX$II5		Illegal interrupt on level 5
22	12	EX$II6		Illegal interrupt on level 6
23	13	EX$II7		Illegal interrupt on level 7
24	14	EX$BTO	*	Bus time-out error
25	15	EX$MMU		MMU error
26	16	EX$PPV	*	Coprocessor protocol violation
27	17	EX$FUB	*	Floating point coprocessor branch/set on unordered condition
30	18	EX$FIR	*	Floating point coprocessor inexact result
31	19	EX$FDZ	*	Floating point coprocessor divide by zero
32	1A	EX$FUN	*	Floating point coprocessor underflow
33	1B	EX$FOE	*	Floating point coprocessor operand error
34	1C	EX$FOV	*	Floating point coprocessor overflow
35	1D	EX$FSN	*	Floating point coprocessor signaling NAN
36	1E	EX$MCE	*	MMU configuration error
37	1F	EX$MIO	*	MMU illegal operation
40	20	EX$MLV	*	MMU access level violation
41	21	EX$FUD	*	Floating point coprocessor unimplemented data type

For the error types flagged with an asterisk, additional information is contained on the stack when the error routine is entered. Otherwise, the stack contains only the status register and PC. Those error types marked with an asterisk format the stack as follows:

@SP	Error trap type
2(SP)	Special status word
4(SP)	Access address
6(SP)	
10(SP)	Instruction register
12(SP)	
14(SP)	Program counter

The "Miscellaneous Exception" error type includes all exception traps currently reserved but unimplemented within the processor itself, unused autovector interrupt vectors, spurious interrupts, and unused TRAP instructions. TRAP15, the breakpoint trap, is also considered a miscellaneous error if the JOBBPT vector contains zero.

Remember on an AM-1000/1200, an access to non-existent memory will be trapped as a parity error, while on VMEbus systems it will be trapped as a bus time-out error. AM-100/L systems ignore accesses to non-existent memory.

JOBWAT - Semaphore Wait Chain Link

RQST and RLSE use this longword field to maintain a chain of JCBs waiting on a particular semaphore. This field contains the JCB address of the next job in this wait chain. Do not modify this field directly.

JOBBPT - The Breakpoint Address

JOBBPT is one longword specifying the address to jump to if a breakpoint (TRAP15 instruction) is encountered during the execution of a user program. The AlphaFIX debugging program uses JOBBPT for breakpoint handling. It is not normally used by other programs.

JOBATT - The Parent Job Index

If the current job is a spawned task, this field points to the JCB of the parent job. Otherwise this field contains zero. Note that the parent job may itself have a parent (and so on), indicated by its JOBATT field being non-zero.

JOBDEV - The Default Device

JOBDEV, one word, 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. This default device is the device your job is currently logged into.

JOBDRV - The Default Drive

JOBDRV, one word, 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. This word is used only 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 you usually use for processing.

JOBTRM - The Terminal Block Pointer

JOBTRM is one longword containing 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 you for modification to set various terminal parameters such as echo control, image mode and lower-case processing. The terminal service routine is described further in Chapter 7 and Appendix B.

JOBRBK - The Run Control Block

JOBRBK, a 28-word area, is the run control block for the job. 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. See the description of the FETCH monitor call in Chapter 4 for more information.

Because the job run block is only a partial DDB, special handling is required inside AMOS each time it is used, making for inefficient operation. Use of the job run block in new software is therefore not recommended.

JOBFPE - The Floating-Point Trap Address

JOBFPE, one longword, contains the address to jump to if an AMOS format floating point error, such as a divide by zero, is executed. A user program which executes AMOS format floating point instructions may enter its error trap address into JOBFPE, or it may use the default error routines which simply display an error message on the user's terminal and exit.

Note this field is only used by the AMOS format (48-bit) floating point routines. IEEE format floating point calls (32-bit and 64-bit) have a different mechanism for error handling. See Chapter 11 for more information.

JOBRNQ - The Scheduling Area

JOBRNQ, a 28-byte area, maintains the parameters for job scheduling and context switching of this job, organized as seven longword pointers and values. The JOBRNQ area is organized as follows:

Octal Location	Hex Location	Meaning
JOBRNQ	JOBRNQ	Link to next runnable job
JOBRNQ+4	JOBRNQ+4	Saved user stack pointer
JOBRNQ+10	JOBRNQ+8	Saved system stack pointer
JOBRNQ+14	JOBRNQ+C	Contains a -1 if job is in run queue, 0 otherwise
JOBRNQ+20	JOBRNQ+10	Job priority (number of jiffies per quanta)
JOBRNQ+24	JOBRNQ+14	Number of jiffies to credit job for next quanta
JOBRNQ+30	JOBRNQ+18	Number of jiffies to charge job for this quanta

Great caution should be used when modifying the JOBRNQ area as any errors made here will most likely cause failure of the whole system.

JOBCPU - The Job's CPU Time Counter

JOBCPU, one longword, contains the number of jiffies the job has spent running. This time does not include IO wait time or other periods when the job is not running, nor does it include time the job was runnable but waiting to be scheduled. See Section 2.1 for information on jiffies and other scheduling information. This field may be used for accounting purposes. Your program can unpack and display this field by using the $OTCPU subroutine described in Appendix D.

JOBCON - The Time and Date the Job Logged-In

One packed longword containing both the time and date the job logged in. This quantity is used to calculate the length of time a job has been logged into the system. This field may be used for accounting purposes.

By subtracting the relevant fields from the current time and date, you can determine the length of time the job has been logged in. Bits 0-16 contain the time of day (in internal format) the job logged in. Bits 17-31 contain the date the job logged in, where the date is stored as the number of days since January 1, 1980. You can convert the packed date to standard internal date format by adding 2444240_{10}. And you can use the $OTCON subroutine described in Appendix D to unpack and display this field.

JOBDSR - The Number of Disk Reads Performed

One longword containing the number of disk reads the job has done since it logged in. The field is incremented by one for every 512-byte disk block requested by a program. Therefore, this field is incremented even if the disk block is returned from DCACHE or Write Cache and not via a physical transfer from the disk. If a request for a logical record spans multiple 512-byte blocks, this field is incremented for every block involved in the transfer. This field may be used for accounting purposes.

JOBDSW - The Number of Disk Writes Performed

One longword containing the number of disk writes the job has done since it logged in. It follows the same rules of incrementing as explained in the JOBDSR section. This field may be used for accounting purposes.

JOBTRC - The Job's Trace Mode Trap Vector

This longword contains the address of a routine to handle traps when executing with the trace bit set in the processor status register. If JOBTRC is zero, the system will display an error message and exit when a trace trap occurs; otherwise, control is transferred to the specified routine. The AlphaFIX debugging program uses JOBTRC; it is not normally used by other programs.

JOBMSR - Reserved

This longword is used internally by the AlphaNET messaging system to track pending messages. Do not modify this field.

JOBFPC - Current Context, Sky Floating Point Board

A 36-byte area used to store the current context of the Sky Fast Floating Point board when it is in use. This area should never be modified, as it may cause the FFP processor to halt the system.

JOBLNG - Point to Current Language Definition Table

This longword contains a pointer to the current language definition table in use by the job. When a job is first created, this field will point to the default language table MONGENed into the system monitor. This field should never be accessed directly. Instead, use the GTLANG monitor call to index the appropriate language definition table for your job, or use SET LANG from AMOS monitor level.

JOBUSN - Current User Name

This twenty byte field contains the current user name for the job. The user name is stored as ASCII characters, terminated by a null, giving a maximum user name length of 19 characters. This field should not be modified. To do so may prevent certain programs and other operating system features from executing properly.

JOBRTP - Current Root PPN

This word field contains the root project-programmer number for the current job. The root PPN is used by various system utilities and should never be modified.

JOBRTD - Current Root Device

This word field contains the root device, packed in RAD50 format, for the current job. The root device is used by various system utilities and should never be modified.

JOBRTU - Current Device Unit Number

This word field contains the device unit number for the current job. The root device is used by various system utilities and should never be modified.

JOBLVL - User Level

This one-byte field contains the "user level" of the user currently associated with this job. The user level, which ranges in value from 0 to 100_{10}, defines the type of functions the user is allowed to perform. A user level of 0 denotes a user with minimum access; 100 denotes a user with no restrictions.

JOBEXP - User Experience Level

This one-byte field contains the "user experience level" of the user currently associated with this job. The experience level, which ranges from 0 to 100_{10}, defines the amount of "help" the user is likely to need. An experience level of 0 assumes a rank beginner, and experience level of 100 assumes an expert.

This field is intended to be used as a guide for how much explanatory information a program gives to the user. For example, in the case of an error message, a user with a lower expertise level might receive a lengthy, detailed explanation of the error and its causes, while an experienced user might just get a notification of the error.

JOBPRM - Current AMOS Command Prompt

This twenty-byte field contains the current AMOS command level prompt associated with the job. The prompt consists of up to 19 ASCII characters terminated by a null byte. This field may be set by the SET PROMPT command from AMOS command level.

For interpreted prompts on systems with PROMPT.SYS installed in system memory, you cannot determine the length of the displayed prompt by counting the characters in JOBPRM. To determine prompt length, follow these steps:

1. There is a longword called RSCPM in the system communications area. If RSCPM is null, PROMPT.SYS is not installed, and you can determine prompt length by scanning JOBPRM.

   If you want your programs to be backwards compatible, you should check the AMOS version (at absolute address 2) for AMOS 2.3(479)-7 or later before interrogating RSCPM.

2. If RSCPM is non-null, call the routine at PH.SIZ+8. offset from the value in RSCPM. On return, register D7 will contain the number of printable characters in the prompt. The routine assumes that dim/bright and foreground color calls do not take up any screen positions.

Programs that use $CMDER to display a caret as a command line error locator should be recompiled with the SYSLIB.LIB file that comes with AMOS 2.3(479)-7 or later. That version of $CMDER is backwards compatible with previous AMOS releases.

For more information about interpreted prompts, see the SET reference sheet in the *AMOS System Commands Reference Manual*.

```
Example code - you may need to modify this - :
        MOV      2, d1          ; check for 2.3(479)-7 or later
        MOV      d1, d7         ;
        AND      #^H0FF000000, d7 ; check VMAJOR \
        CMP      d7, #2_24.     ;
        BHI      10$            ;
        BLO      20$            ;
        MOV      d1, d7         ;
        AND      #^H0F0000, d7 ; check VMINOR
        CMP      d7, #3_16.     ;
        BHI      10$            ;
        BLO      20$            ;
        MOV      d1, d7         ;
        AND      #^H0FFF, d7    ; check VEDIT
        CMP      d7, #479.      ;
        BHI      10$            ;
        BLO      20$            ;
        MOV      d1, d7         ;
        AND      #^H0F00000, d7  ; check VWHO
        CMP      d7, #7_20.     ;
        BLO      20$            ;
10$:    MOV      RSCPM, d7      ; get prompt install vector
        BEQ      20$            ; not there, br
        MOV      d7, a6         ;
        CALL     PH.SIZ+8.(a6) ; call length function, len into d7
        BR       40$            ;
20$:    JOBIDX   a6             ; index jcb
        CLR      d7             ; clear length count
        LEA      a6, JOBPRM(a6)  ; index the job prompt.
30$:    TSTB     (a6)+          ; Null yet ?
        BEQ      40$            ;  Yes.
        INC      d7             ;  No. Bump counter.
        BR       30$            ;  Again.
40$:    ....                    ; length of prompt in d7
```

JOBCMD - Default Command Line

This thirty-byte field contains the default command line executed whenever the job returns to AMOS command level. The field can contain up to 29 ASCII characters, terminated by a null byte.

When a job is first created, this field contains a null, meaning no command is forced, but instead AMOS waits for the user to enter a command from the terminal keyboard. This is the normal mode of operation.

In certain applications, however, it is useful to be able to force a user to stay within a specific program. To do this, enter the name of the program, or any other valid AMOS command line of 29 or fewer characters, into the JOBCMD field. Once this has been done, any time the job reaches AMOS command level, the command within JOBCMD is automatically executed, just as if it had been entered from the keyboard. This technique forces a user to remain within the program, regardless of what action might be taken. For example, the following code would force the user to stay within the AlphaMENU shell:

```
JOBIDX   A0             ; index our JCB
LEA      A6,JOBCMD(A0) ; index the command buffer
MOVB     #'S,(A6)+     ; transfer the command name
MOVB     #'H,(A6)+
MOVB     #'E,(A6)+
MOVB     #'L,(A6)+
MOVB     #'L,(A6)+
CLRB     @A6            ; terminate the command
```

To once again allow the user to return to AMOS command level and stay there, simply clear the first byte of the JOBCMD field. For example:

```
JOBIDX   A0             ; index the JCB
CLRB     JOBCMD(A0)     ; return to null command mode
```

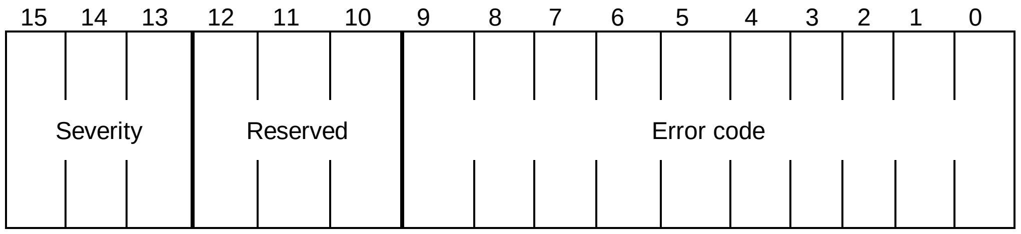
JOBDSC - The Job's DSECT Pointer

This entry is used to index the job's currently active DSECT area. This pointer is used by various language processors (such as AlphaC) to keep track of the data area and to allow that data area to be located by other programs and subroutines.

JOBERR - Job Error Value

This 16-bit field is used to record the result of the previously executed program. By examining this field, a program can determine whether or not the previous program ran to successful completion, or was aborted due to an error. It is this field the IF program looks at to check for execution errors.

The JOBERR field is itself broken down into two separate fields of three and ten bits. The remaining three bits are reserved for future expansion. The first group of three bits is used for a severity code. The ten bit field is used to contain the error specific code. The JOBERR word looks like:



The severity field contains a value describing the severity of the error, ranging from 0 to 17 octal where the lower the number the more severe the error. The SEVERITY function in the IF program returns a logically inverted value allowing simpler numeric comparison of the severity field (higher number indicates greater severity).

The assigned severity values are as follows:

Value	Meaning
0	If error code is nonzero AND severity code is zero, fatal error: operation aborted due to severe error.
0	If error code is zero AND severity code is zero, operation completed with no errors or warnings.
2	Operation aborted due to error or ^C.
4	Operation completed with error(s).
6	Operation completed with warning(s).

If the entire word is 0—both the severity and the error code fields—the job completed with no errors or warnings. If the severity value is 0 and the error code is non-zero, it indicates a fatal error. Note it is possible to have a non-zero severity field with a zero error code, such as when only warnings were issued. The defined error status values are:

Octal Value	Hex Value	Symbol	Meaning
000000	0		No error has been detected
000001 thru 000377	1 – FF		Standard DDB error codes. See Chapter 6
000400	100	JE$MSC	Miscellaneous error (catch-all error number)
000401	101	JE$MAP	Memory map destroyed
000402	102	JE$IPR	Insufficient privileges to run program
000403	103	JE$L12	Must be logged into [1,2] to run program
000404	104	JE$OPR	Must be logged into DSK0:[1,2] to run program
000405	105	JE$M20	Program requires 68020 processor
000406	106	JE$LOG	Must be logged in to run program
000407	107	JE$BER	Bus error
000410	108	JE$PAR	Memory parity error
000411	109	JE$ADR	Address error
000412	10A	JE$IIN	Illegal instruction
000413	10B	JE$DIV	Divide by zero
000414	10C	JE$CHK	CHK instruction trap
000415	10D	JE$TRP	TRAPV instruction trap
000416	10E	JE$PRI	Privilege violation
000417	10F	JE$TRC	Trace trap return
000420	110	JE$EM1	EM1111 instruction trap
000421	111	JE$MEX	Miscellaneous exceptions
000422	112	JE$II0	Illegal interrupt on level 0
000423	113	JE$II1	Illegal interrupt on level 1
000424	114	JE$II2	Illegal interrupt on level 2
000425	115	JE$II3	Illegal interrupt on level 3
000426	116	JE$II4	Illegal interrupt on level 4
000427	117	JE$II5	Illegal interrupt on level 5
000430	118	JE$II6	Illegal interrupt on level 6
000431	119	JE$II7	Illegal interrupt on level 7
000432	11A	JE$BTO	Bus time-out error
000433	11B	JE$MMU	MMU error
000444	124	JE$PPV	Coprocessor protocol violation
000445	125	JE$FUB	Floating point coprocessor branch/set on unordered condition
000446	126	JE$FIR	Floating point coprocessor inexact result
000447	127	JE$FDZ	Floating point coprocessor divide by zero
000450	128	JE$FUN	Floating point coprocessor underflow
000451	129	JE$FOE	Floating point coprocessor operand error
000452	12A	JE$FOV	Floating point coprocessor overflow
000453	12B	JE$FSN	Floating point coprocessor signaling NAN
000454	12C	JE$MCE	MMU configuration error
000455	12D	JE$MIO	MMU illegal operation
000456	12E	JE$MLV	MMU access level violation
000457	12F	JE$FUD	Floating point coprocessor unimplemented data type
000600	180	JE$LPA	Language processor aborted
040601	4181	JE$LAE	Language processor aborted with errors
100602	8192	JE$LCE	Language processor completed with errors
100603	8193	JE$UND	Undefined identifier in input
100604	8194	JE$RIE	Runtime interpreter error
100605	8195	JE$SYN	Syntax error in input
100606	8196	JE$AEL	Assembly error in linkage
041000	4200	JE$CTC	Process aborted by operator (^C)
001001	201	JE$CLF	Command line format error
001002	202	JE$CLS	Command line switch error
001003	203	JE$SSD	Bad SSD
001004	204	JE$ONF	Overlay not found
140000	C000	JE$WRN	Warnings were issued

Normally AMOS clears the JOBERR status word before executing a program, thus obliterating the status returned by the previous program in a command file. To allow a program to have access to the error status word left over from the previous program, a bit in the program header word must be set. By setting PH$ERR within the program header (PHDR), a program may tell AMOS to leave the error status word intact. Any program wishing to examine the status word must have this bit set.

Once the PH$ERR bit is set, a program may simply access the JOBERR word within the job control block and take appropriate action based on its value. See the section on Program Header Format in Appendix A for further details.

JOBDFP - Default File Protection

This 32-bit field contains the file protection to be assigned to any new files created by this job. This field is set up by LOG or LOGON, both of which read the desired setting for the default file protection from the USER.SYS file. The values in this field correspond to the values defined in the D.PROT field described in Chapter 6.

This field should not be changed directly. If a user wants to set a different default file protection, the USER.SYS entry for that user should be changed by using the MUSER program.

JOBFCB - Floating Point Coprocessor Control Block

This field is used to store the current state of the floating point co-processor during job scheduling context switches. Because its contents vary depending on the specific hardware (or software) in use, it contents should never be relied upon.

JOBFCP - Floating Point Coprocessor Context Pointer

This longword contains a pointer into the JOBFCB area, allowing the job scheduler to keep track of the size of the currently stored floating point co-processor context.

JOBSIV - Software Interrupt Vector Table Pointer

This longword contains a pointer to the currently active software interrupt vector table. If no software interrupts are enabled, this field will be zero.

This field should never be manipulated directly. Instead, use the software interrupt monitor calls described in Chapter 14.

JOBSIM - Software Interrupt Enable Mask

This 32-bit field contains a bit mask defining the currently enabled software interrupt levels. If no software interrupts are enabled, this field will be zero.

This field should never be manipulated directly. Instead, use the software interrupt monitor calls described in Chapter 14.

JOBSIP - Software Interrupt Pending Mask

This 32-bit field contains a bit mask defining any pending software interrupts. If no software interrupts are pending, this field will be zero.

This field should never be manipulated directly. Instead, use the software interrupt monitor calls described in Chapter 14.

JOBSIT - Software Interrupt Timer Pointer

This longword contains a pointer to the software interrupt timer data structure.

This field should never be manipulated directly. Instead, use the software interrupt monitor calls described in Chapter 14.

JOBERS - Error Context Save Area

This longword contains a pointer the job's previous error context. Used by the STDERR monitor call. This field should never be directly accessed or modified.

JOBPLK - PLOCK Nesting Count

This longword contains a count of the number of times this job has executed a PLOCK monitor call without a corresponding PUNLOK. This count is used to maintain the nesting capability of PLOCK. This field should never be modified directly; the EXIT monitor call will clear this field if proper nesting is not performed.

JOBIEE - IEEE Floating Point Error Vector

This longword contains a pointer to the job's error handling routine for IEEE floating point arithmetic. When a floating point error occurs, control will be transferred to this routine, as described in Chapter 11.

JOBESP - Pointer for Screen Processor

This longword contains a pointer used by the ESP screen processor software. This field should never be manipulated directly.

JOBRFU - VDK/USAM Impure Pointer

This longword contains a pointer to the job's impure area used by the VDK virtual disk and the USAM access method. This field should never be manipulated directly.

JOBCOF - Current Open Object File Pointer

This longword contains a pointer to a data structure defining the currently open object files for this job. This field should never be manipulated directly.

JOBROF - Root Object File Handle

This longword contains the handle of the root object file for this job. This field should never be manipulated directly.

JOBRMF - Network List

This longword contains a pointer to a linked list containing all surrogate networks. This field is used internally by AlphaNET and should not be modified.

JOBNTB - RPC Buffer Pointer

This longword contains a pointer to the buffer used by the Remote Procedure Call service of AlphaNET. This field is used internally by AMOS and should not be modified.

JOBRES - Resource Manager Queue Pointer

This longword contains a pointer to a queue used by AMOS' resource manager. This field is used internally by AMOS and should not be modified.

JOBTSP - Network Transport Service Buffer Pointer

This longword contains a pointer to the buffer used by the Transport Layer service of AlphaNET. This field is used internally by AlphaNET and should not be modified.

JOBFCB - Hardware Floating Point Context

This field is used to store the current context of the floating point coprocessor (68881 or 68882) during job context switching. It should never be modified.

JOBSIS - Terminal Output Software Interrupt Structure

This longword contains a pointer to the software interrupt structure used by this job for terminal output buffering. This field is used internally by AMOS and should never be modified.

JOBSSP - The Job's Supervisor Stack Area

JOBSSP is a 600-word area that acts as the supervisor stack for this job. SP is set to the top of this area when the job is initialized. The job scheduler saves all user registers and processor status on the supervisor stack during job context switching, monitor calls, and interrupts.

JOBUSP - The Job's User Stack Area

JOBUSP is a 400-word area that acts as the user stack for this job. SP is set to the top of this area when a new program is initiated. You may set your stack pointer to use an area other than JOBUSP by moving the address of a larger area within your own partition to SP, if the program needs more stack area.

Chapter 3
Memory Control System Calls

To optimally use the memory resources available on the Alpha Micro computer and minimize system problems due to memory access violations, the assembly language programmer should understand how AMOS allocates memory and the rules for accessing memory. This section describes the memory allocation scheme and the monitor calls that assist you in using memory in the proper way.

The MC68000- and MC68010-based Alpha Micro computer systems can address up to a total of 16 megabytes of memory. The MC68020-based and later systems can address as much as four gigabytes. The AMOS monitor resides in low memory beginning at location zero and extending upward as far as the monitor requires (typically around 120K 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 system resources and user memory partitions for each of the jobs.

You may allocate all of the user memory to one job, or split it up into several partitions of varying sizes allocating one partition to each job. The amount of memory a user program has available is, therefore, defined as the single contiguous memory partition assigned to a job by the MEMORY command.

AMOS then allocates this memory partition block into smaller defined blocks called "modules" which you and the system can use to contain programs and data areas. There are monitor calls that allow your programs to locate the absolute boundaries of its own memory partition and also to allocate, change, and delete memory segments in the form of defined modules. You can name these modules just like files (filename.extension), so you can locate them 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.

MEMORY PARTITION FORMAT

The memory partition assigned to a job may be located anywhere in memory depending on the memory available when the job assigned memory to itself using the MEMORY command. Your programs should not try to anticipate any specific location for this partition. Within the partition, AMOS allocates memory modules beginning at the base of the defined partition and building modules upward on top of each other as far as space permits. The system will not build modules that extend past the top boundary of the partition. As modules are deleted from memory, all modules above them automatically shift downward to fill up the space the deleted module left. Also, when the size of any module changes, the modules above it shift in position accordingly. This practice insures all available memory is always at the top of your partition in one contiguous block. And because the system uses the first portion of free memory for loading a program, you must write all your programs in totally relocatable code.

Figure 3-1 shows a typical memory layout for three users operating in a 4MB system. The free memory at the 3600K boundary could be used by a fourth job or by the third job if its partition were expanded.

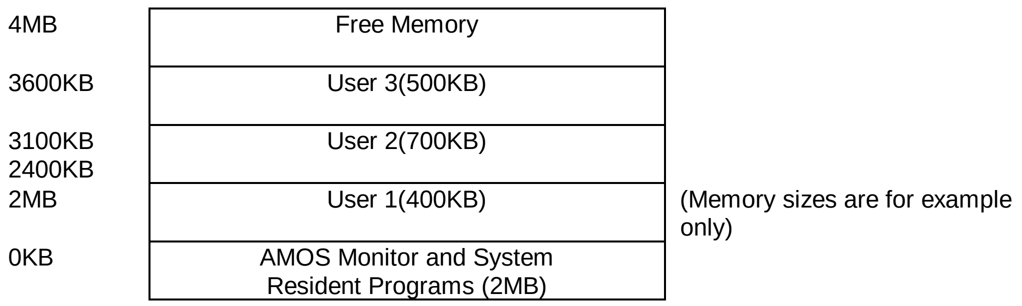

4MB
Free Memory
3600KB
User 3(500KB)
3100KB
User 2(700KB)
2400KB
2MB
User 1(400KB)
(Memory sizes are for example only)
0KB
AMOS Monitor and System Resident Programs (2MB)

Figure 3-1: Memory Map for a Typical 4MB System

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

Call	Meaning
USRBAS arg	Absolute base of user memory partition (first word)
USREND arg	Absolute end of user memory partition (last word)
USRFRE arg	Current base of remaining free memory (end of last module+2)

Since modules must always occupy an even number of bytes, the above calls always return an even address. If no modules are allocated in the current partition, the USRFRE address equals the USRBAS address. Otherwise, the USRFRE address is the word following the last currently allocated module in the memory partition. You can calculate the remaining free user memory module which it may use for storage of symbols or some similar function. Then it opens two I/O files on disk which causes 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 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 being processed, 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. Using the USREND call insures the program will use the free memory without having to compensate for the remaining part of any command file module.

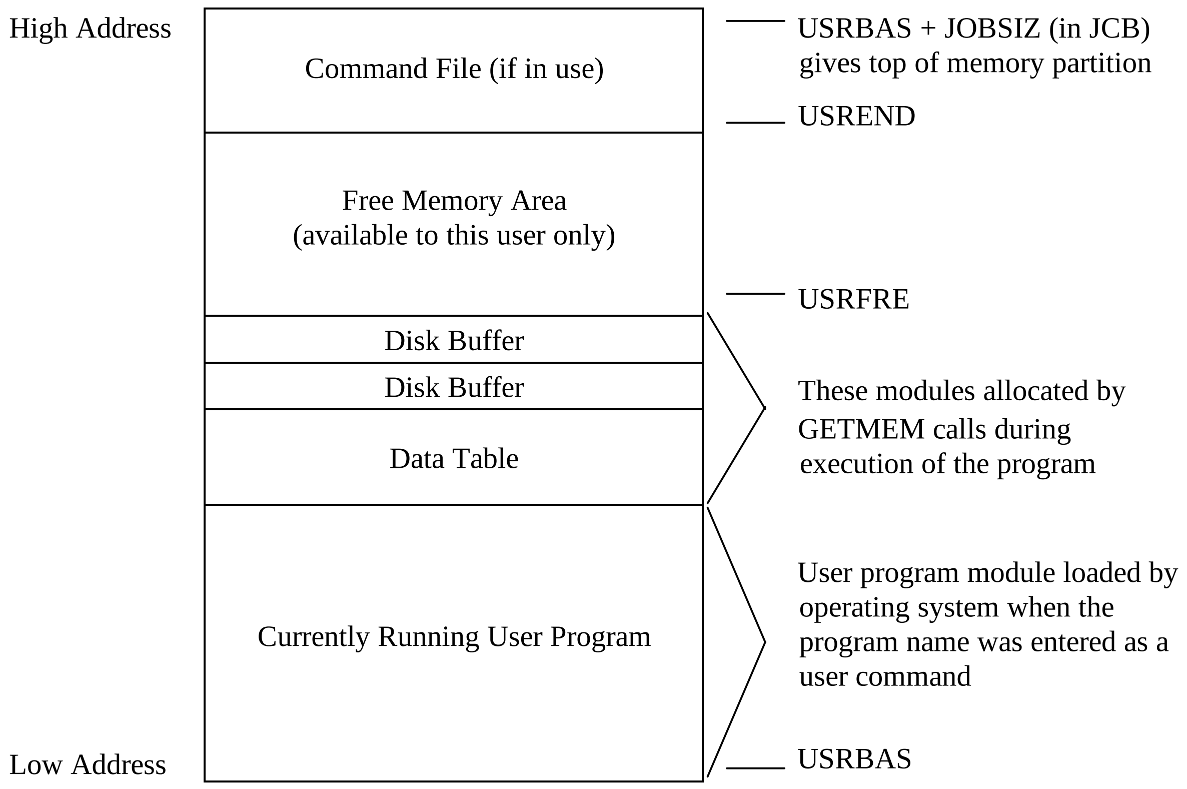

High Address
Low Address
Command File (if in use)
Free Memory Area
(available to this user only)
Disk Buffer
Disk Buffer
Data Table
Currently Running User Program
USRBAS + JOBSIZ (in JCB) gives top of memory partition
USREND
USRFRE
These modules allocated by GETMEM calls during execution of the program
User program module loaded by operating system when the program name was entered as a user command
USRBAS

Figure 3-2: Memory Map for a Typical User Job Partition

The standard use of memory by the operating system is through the use of the memory management system calls, to be described in the next section. However, it is theoretically possible to use 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. However, we do not recommend using this technique due to its potential incompatibility with other software on the system. In particular, note that initializing files normally results in the allocation of a buffer module; the operating system allocates this buffer at the current setting of the USRFRE address, then updates that USRFRE address.

Therefore, you 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.

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 automatically pad an odd-sized module with a null byte to even it up. All modules contain six housekeeping entries followed by any number of data words from zero to the maximum size left in the user memory partition. The six housekeeping words are always allocated, so a single-word module really takes up eight words of memory. The size of the housekeeping area has been given the symbol ZID in SYS.UNV.

The module format is as follows:

Word Offset From Start	Meaning
1-2	Total module size in bytes including housekeeping words
3	Module flag word
4-5	Module filename packed RAD50
6	Module extension packed RAD50
7 thru n	Module data area

Figure 3-3 illustrates this standard module format from another perspective. The data area is usually the only area that concerns the user, and so all references are made from the base of this area. The SRCH and FETCH calls, described in section 4.1, 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 by negative offsets relative to the data base address.

When scanning for a specific module or locating the end of the current module string, you may set your index using the USRBAS call, which returns the address of the size longword of the first allocated module. You can 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 longword to the index. This bumps the index to the next module allocated. The last module always has a zero longword following it, and you must be careful not to destroy this zero longword if you are manipulating free memory directly without allocating it using the memory calls.

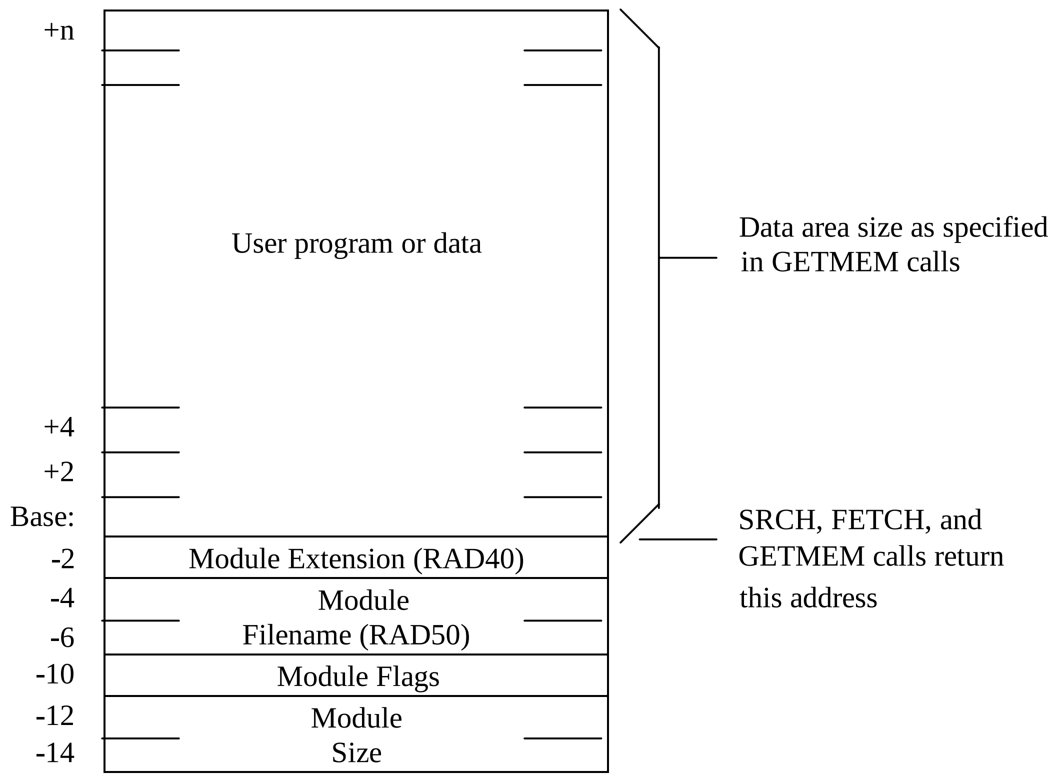


Figure 3-3: Standard Memory Module Format

The module filename and extension follow the same format as the filenames on disk if the module in memory is named. The name is optional and need only be used 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. The monitor automatically deletes temporary modules 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 it also deletes all modules above it (zero is the module area termination longword word).

The module flags word is defined as follows:

Symbo l	Octal Value	Hex Value	Meaning
FIL	2	2	This module is permanent. Don't delete it when an EXIT call is executed.
FGD	4	4	This is a foreground module. This flag is reserved for future use.
OBM	10	8	This module has an odd byte count. The file on disk from which the module was loaded was an odd number of bytes in size. The memory module size is rounded up to the next even number.
LOK	20	10	This module is locked in memory. Don't allow it to be deleted via the DELMEM call or the DELETE program

MANIPULATING MEMORY MODULES

Three monitor calls create, alter and delete memory modules. All three calls take a single standard argument which must be the address of a 4-word block called a memory control block (MCB). The first longword of this MCB contains the absolute memory address of the data area in the allocated module (past the housekeeping words). The second longword contains the size of the data area in bytes (twelve 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. You need not be concerned with the housekeeping words unless you need to access them directly. While the MCB can be allocated as a fixed set of memory locations, it is usually more convenient to allocate them temporarily on the stack, as shown below.

The following three calls manipulate memory modules:

Call	Purpose
GETMEM adr	Allocates a new memory module at current USRFRE, returns address in ***adr***.
CHGMEM adr	Changes the size of the module defined by MCB indexed by ***adr***.
DELMEM adr	Deletes the memory module defined by MCB indexed by ***adr***.

The Z-flag is reset if GETMEM and/or CHGMEM fail (i.e., there is insufficient memory). ***It is vital that this flag be checked after every call, as ignoring a failing memory allocation may cause your program to use memory already allocated to another job, potentially causing system failure.***

Allocating a Memory Module

The following example shows the allocation of a 100-byte module:

```
               PUSH     #100.         ; set module size as 100 (decimal) bytes
               PUSH                   ; reserve space for module address
               GETMEM   @SP           ; allocate module (@SP gets its address)
               BNE      NOMEM         ; no memory available -
               POP      index         ; remove address of allocated module
               POP                    ; remove module size
               ...
               ...
       NOMEM:  EXIT
```

Changing a Memory Module

You may increase the size of the same module by:

```
               PUSH     #120.         ; change size to 120 (decimal) bytes
               PUSH     index         ; store the address of the module
               CHGMEM   @SP           ; change its size
               BNE      NOMEM         ; not enough memory available -
               POP                    ; remove the temporary MCB from the stack
               POP
```

The above code causes the monitor to adjust the module housekeeping size word to reflect the new size. The address of the module does not change.

However, note the USRFRE address advances by 20 bytes and any modules allocated after the one at MCB shift up in memory; but the monitor does not adjust their corresponding addresses in their MCB. I/O buffers allocated after the MCB module will, therefore, be erroneously addressed after the change, so use the CHGMEM call with care.

Deleting a Memory Module

To delete the above module, we use the code:

```
               PUSH                   ; push dummy size onto stack
               PUSH     index         ; push address of module onto stack
               DELMEM   @SP           ; delete the module
               BNE      NOMEM         ; failure -
               POP                    ; remove temporary MCB from stack
               POP
```

PERMANENT AND TEMPORARY MODULES

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

```
        PUSH     #200.         ; set size as 200 bytes
        PUSH                   ; reserve space for index
        GETMEM   @SP           ; allocate the module
        BNE     NOMEM          ; no memory available -
        POP     A5             ; A5 will index the module
        POP                    ; remove size
        MOV      A5,A0         ; set A0 to index the data area base
        MOVW     #[DAT],-(A0)  ; set module name and extension (RAD50)
        MOVW     #[LE1],-(A0)  ;   into the housekeeping words
        MOVW     #[TAB],-(A0)  ;   in reverse for efficient use of A0
        ORW      #FIL,-(A0)    ; set permanent file bit on in flag word
```

You may save permanent memory modules onto disk using the operator SAVE command, or you may delete them from memory when done with the operator DEL command. See the *System Commands Reference Manual* for details on these commands.

ALLOCATING MODULES WITH GETIMP

One of the most frequent uses of the memory module allocating calls is allocating a memory area for impure storage. The GETIMP macro makes this and other simple memory allocation operations easier. To use GETIMP simply specify:

```
        GETIMP   size, index, {error-routine}
```

In this example, ***size*** specifies the number of bytes to be allocated, ***index*** is any valid destination argument that specifies where the base of the new memory module is to be placed, and ***error-routine*** is an optional argument that specifies the routine to execute if there is not enough memory available for the requested allocation. If you don't specify error-routine, the operating system simply prints an error message and aborts.

For example, to allocate a 100 (decimal) byte impure module and index it with A5, using the default error routine, you would enter:

```
        GETIMP   100.,A5
```

Chapter 4
Allocating and Using Memory

This chapter discusses how to allocate and use memory modules, and also discusses the Shared Memory Facility, which allows multiple jobs and programs to share a memory pool allocated at bootup time.

MEMORY MODULES - SRCH AND FETCH CALLS

Memory modules may contain an optional filename and extension, which you can use to locate them, both in memory and on the disk. This chapter explains locating and loading modules using these optional filenames and extensions. Normally, when you enter a command from the terminal, AMOS first searches for the requested program in the resident system memory area, then in your own memory partition. If the program is resident in either of these places, it does not need to be loaded in from disk, and execution begins immediately.

You may use two monitor calls, FETCH and SRCH, for locating and loading modules in memory by name. Actually, the SRCH call is a specialized version of the FETCH call and is included only for convenience. Basically, the SRCH call only locates a module if it is already in memory, while the FETCH call automatically loads a module into memory from the disk if it is not found in memory. Both calls have the same basic format:

```
SRCH     nameblock adr,index dst,control-flags src
FETCH    nameblock adr,index dst,control-flags src
```

Specifying the Module Name

nameblock is a standard argument used in the SRCH and FETCH calls to specify the name of the desired module. The format of nameblock referenced is different in each case, however. In the case of the SRCH call, nameblock refers to a 3-word block of memory containing the filename and extension of the module you want in RAD50 packed form. For the FETCH call, nameblock refers to a Dataset Driver Block (DDB) which allows you to specify a full disk file specification to use in loading the module from disk in case it is not already in memory. In brief, the DDB is an 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. See Chapter 6 for more information.

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. See Figure 3-3 in the preceding chapter for the layout of the memory module and the place this index is set to. Index is a standard argument, although the normal mode is to receive the module address in an address register (A0-A6).

Flags

The third argument is the optional control flags, which you may use to control the operation of the SRCH and FETCH calls. This argument is any valid expression with a value in the range of 0-17 (octal). Only the low order five bits are significant, and they have the following mnemonic definitions in the system library SYS.UNV:

Symbol	Octal Value	Hex Value	Meaning
F.FCH	1	1	Fetch module from disk if not in memory
F.USR	2	2	Search user memory only
F.ABS	4	4	Load absolute segment from disk
F.FIL	10	8	Set module permanent file flag after load from disk
F.NFCH	20	10	Force "no fetch from disk," overrides F.FCH

Note that if the index destination (the second argument) is not an address register, these flags will ***NOT*** be set properly.

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 monitor call. The SRCH call forces this bit off while the FETCH call forces this bit on. When set, the F.FCH bit causes the program to interpret nameblock as a full file DDB and to load the module from disk if it cannot be located in memory first. Since the use of this bit is controlled by specifying either SRCH or FETCH as the calling opcode, you should not include this bit in the control-flags argument of your call.

F.USR - Search User Memory Only

F.USR tells SRCH and FETCH to bypass searching the resident system memory area for the module and proceed directly to searching the user area only. This allows the program to load and use specific versions of its modules even though they may be duplicated in the system memory area. Normally, only system software uses this flag.

F.ABS - Bypass Memory Search

When set, F.ABS forces a direct search to the disk for the requested module, bypassing all memory searches that would normally occur. The module then loads into memory at the absolute address by the buffer address (D.BUF) in the specified DDB. It allocates no housekeeping words, and the first word of the module gets loaded into the first word specified by the buffer address. You can use the F.ABS form of the FETCH call to load program segment overlays.

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. Do not use this option if you specify F.ABS. The FETCH call always places the filename and extension into the housekeeping words; so even if the module is only temporary, it can still be located by name, as long as the program that 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 the operating system will not delete the module from memory when the program exits. The LOAD program uses this method to load a program into memory and leave it there after execution is completed.

Completion Codes

When the SRCH or FETCH call returns, your program 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.

In addition to the Z-bit being reset in case of an error, the DDB used by a FETCH call will contain the error code in the D.ERR status byte.

Examples

The following examples illustrate the use of the SRCH and FETCH monitor calls.

Locating a Memory Module

The code shown below searches the system resident area and then the user's memory partition for the module named "TABLE.DAT." If it finds the module, it returns the address in A1. If not, it displays an error message.

```
        SRCH     NAMSTR,A1     ; try to find TABLE.DAT
        BNE      ERROR         ;   not found - process error
                               ; Address of TABLE.DAT is now in A1
        ...
ERROR:  TYPECR   <?TABLE.DAT not found in memory>
        EXIT
NAMSTR: RAD50    /TABLE DAT/
```

Loading an Overlay

The program segment shown below loads a program overlay into memory. It uses the job run block (JOBRBK) to load the overlay module from disk. This example assumes the symbol OVRBAS has been defined as the base of the overlay area through use of the OVRLAY pseudo-op within the source program. The example tries to load the overlay module "OVRMOD.OVR" into memory. If it can't find the module, it displays an error message.

```
JOBIDX   A0             ; index our own JCB
LEA      A0,JOBRBK(A0) ; index our job run block
CLRW     D.DEV(A0)     ;  set for default drive
MOVW     #-1, D.DRV(A0)
MOVW     #[OVR],D.FIL(A0); set filename to OVRMOD.OVR
MOVW     #[MOD],D.FIL+2(A0)
MOVW     #[OVR],D.EXT(A0)
CLRW     D.PPN(A0)
LEA      A1,OVRBAS     ; index the overlay area
MOV      A1,D.BUF(A0)  ; store in buffer address
FETCH    @A0,A6,F.ABS  ; load it in (note use of F.ABS)
BEQ      OVRBAS        ; loaded ok - go execute overlay
TYPE     <?>           ; not found - display error
PFILE    D.FIL(A0)     ; display filename
TYPECR   < not found>
EXIT
```

For more information on overlays, see the *AMOS Assembly Language Programmer's Manual*.

SHARED MEMORY FACILITY

The AMOS Shared Memory Facility allows you to define a memory pool at bootup time via the SMEM command in the system initialization command file. Multiple jobs and programs can then make use of this memory pool, allocating and using blocks of memory. Use of these blocks of memory can be coordinated by locking and unlocking them.

GETSHM - Get/Search Shared Memory

GETSHM gets a block of shared memory. The format of the call is:

```
GETSHM  {name},{size},{adr},{shmid},{SM$PRM}
```

name	is the address of a six-character/RAD50 name by which to identify the memory block.
Size	is the size of memory desired, in bytes.
Adr	is where the address of the memory block will be stored.
Shmid	is where the shared memory ID is to be stored.
SM$PRM	indicates that this memory block is to be allocated permanently to the requester.

The call returns:

A6	Address of the memory block.
Adr	If specified, stored in A6.
D6	Actual size allocated/found.
D1	Shared Memory ID.
Shmid	If specified, stored in D1.
D7	Destroyed.

Condition codes are:

Z	1 for successful operations (use BEQ).
C	1 if already defined (use BCS).
Z	0 for unsuccessful operations (use BNE).

GETSHM is used to get a block of shared memory. Name is the address of a six-character, RAD50 name used to identify the memory block.

If name is specified and the block already exists, its address will be returned, its use-count will be incremented, the actual size of the block will be returned in D6, and both the Z and C flags will be set. If the block does not exist, it will be allocated and given name specified.

Name may also be used in subsequent DELSHM calls to identify the memory block to delete.

If name is not specified, the memory block allocated will be unnamed.

Memory allocated to a job will be recorded in the shared memory queue and on exit, any unreleased memory blocks will be returned by the operating system to free memory. A Shared Memory ID (Shmid) is also returned for use with the LOKSHM and UNLKSHM calls.

If size is not available, the operation will be considered unsuccessful, and the Z flag will be reset (set to zero). If the memory block already exists, but its size is larger than that which was allocated, the actual size will be returned in D6 and the C flag will be set, and the Z flag will be reset. If size is omitted, it is assumed that your program has set the size in D6.

Adr represents the location at which to store the address of the shared memory block. If omitted, the address will simply be left in A6.

shmid represents the location at which to store the Shared Memory ID of the memory block. This ID is used to identify the shared memory block in subsequent LOKSHM and UNLKSHM calls. If omitted, the ID will simply be left in D1.

SH$PRM indicates that the request is allocating shared memory on a more or less permanent basis, that a "job context" may not be present, and that the requester will insure that the memory block is returned to free memory, if desired, via the DELSHM call.

See the sections on the DELSHM, LOKSHM, and UNLKSHM calls for more information on the Shared Memory Facility.

Since shared memory is global by nature, there is nothing to keep your program from trashing the system, since all references are to pointers.

You can allocate shared memory at any time a monitor call is issued.

EXAMPLE:

Allocate a 1024-byte block of shared memory permanently, with the name FLPBUF and store its address in A1. Store the Shared Memory ID in the location called SID.

```
10$:    LEA      A5,START      ; Start of program
                                ; Allocate memory.
        GETSHM   NAM,#1024.,A1,SID(A5),SM$PRM
        BEQ      OK            ; Branch if OK.
        SLEEP    #10.          ; -- error, sleep a bit.
        BR       10$           ; go try again.
        ...
OK:     ...                    ; OK, A1 has memory address.
        ...
NAM:    RAD50    /FLPBUF/
SID:    LWORD    0
```

DELSHM - Release Shared Memory

DELSHM releases a block of shared memory. The format of the call is:

```
DELSHM  {name},{adr},{SM$ALL}
```

name	is the address of a six-character/RAD50 name used to identify the memory block.
Adr	is the address of the shared memory block being released.
SM$ALL	means "return all memory allocated."

Condition codes are:

Z	1 for successful operations (use BEQ).
Z	0 for unsuccessful operations (use BNE).

DELSHM releases a block of shared memory obtained via GETSHM.

name is the address of a six-character RAD50 name used in the GETSHM call to identify the memory block. If the name is omitted, the memory block must be specified by adr, unless SM$ALL is specified, in which case the address parameter is ignored.

adr is the address of the shared memory block to release. If omitted, the address is assumed to be in A6, unless name is specified, in which case adr will be ignored. If SM$ALL is specified, however, the address parameter is ignored.

SM$ALL indicates that your program wants to release all shared memory thus far allocated. This option is only meaningful if there is a job context under which shared memory was previously allocated (i.e., the SM$PRM option was ***not*** specified in the GETSHM calls).

An error will be returned if the shared memory block specified does not exist. If several requests have used the GETSHM call to access the block, only the last DELSHM call will actually deallocate the specified block. When a job exits, any unreleased memory blocks will be returned by the operating system back to the free pool, unless the SM$PRM flag is set (see the section on GETSHM, above).

For more information on the Shared Memory Facility, see the sections in this chapter on GETSHM, LOKSHM, and UNLKSHM.

EXAMPLE:

Release the 1024-byte block of shared memory allocated in the example in "GETSHM," above.

```
10$:    DELSHM   NAM          ; Release the block named FLPBUF.
        BEQ      OK           ; Branch if OK.
        ...                   ; -- error.
OK:     ...
        ...
NAM:    RAD50    /FLPBUF/
```

LOKSHM and UNLKSHM - Lock and Unlock Shared Memory

LOKSHM and UNLOKSHM are used to coordinate use of shared memory by locking and unlocking a specific shared memory block.

The call format is:

```
LOKSHM  {shmid}{,NOWAIT}
```

and:

```
UNLKSHM {shmid}
```

shmid	is the address of the Shared Memory ID to lock or unlock.
NOWAIT	means "don't wait if I cannot get the memory block."

Condition codes are:

Z	1 for successful operations (use BEQ).
Z	0 for unsuccessful operations (use BNE).

LOKSHM sets a semaphore located in the Shared Memory ID, and UNLKSHM clears it. With the LOKSHM call, a parameter NOWAIT may be specified and LOKSHM will return immediately, regardless of whether or not the semaphore was successfully obtained. Your program can test success by a BEQ; if Z is zero, the semaphore was not obtained.

shmid is the address of the Shared Memory ID obtained with the GETSHM call. If omitted, it is assumed your program pre-loaded it into A6.

NOWAIT indicates your program will test the success of the semaphore access. If omitted, LOKSHM will wait until the semaphore is granted. This parameter is only used with LOKSHM. Also, there is no significance to the value put in this parameter position; if it is non-blank, NOWAIT is assumed. There is no WAIT keyword; if the NOWAIT position is blank, WAIT is implied.

See the sections on GETSHM and DELSHM in this chapter for more information on the Shared Memory Facility.

GETSHM and DELSHM do not check the semaphore used by LOKSHM and UNLKSHM. LOKSHM and UNLKSHM are simply for your program's convenience.

EXAMPLE:

Lock the previously allocated block of memory whose Shared Memory ID is in SID. Sleep for awhile if the semaphore cannot be granted.

```
10$:    LOKSHM   SID(A5),NOWAIT   ; Try to get the semaphore.
        BEQ      OK              ; Branch if OK.
        SLEEP    #10.            ; -- Couldn't get it, so sleep.
        BR       10$             ; Go try again.
OK:     ...                      ; OK.
        ...
SID:    LWORD    0
```

Chapter 5 Monitor Queue System Calls

AMOS provides a general purpose queue system that is used by various internal monitor routines and some system programs. Because the queue system is an integral part of AMOS, and because the system cannot run without sufficient queue blocks, we don't recommend using the queue system within user programs. However, we document it here to give you an understanding of the way the queue system functions.

The queue is a list of data blocks linked to each other in a forward linked list. The base of this list, and the count of the blocks in the list, are contained in the QFREE monitor communications words (see Appendix C).

Each queue block in the list links to the next one by storing the address of the next block in the first longword 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 9 longwords (36 bytes) in size. The monitor initially contains a small number of blocks in the available queue list, sufficient to boot the system and start processing the system initialization command file. Additional queue blocks may be reserved by using the QUEUE initialization command (see the section on queue block use, below).

During normal monitor operation, various functions use these queue blocks to do certain tasks. When a routine needs a queue block, it issues a QGET monitor call, which delivers the first available queue block by returning its base address. The routine then uses this area to temporarily store information during processing. When the routine no longer requires the block, it issues a QRET monitor call, which returns the queue block to the available list for later use.

The monitor queue system is used to provide storage for interrupt-driven hardware, such as disk and terminal controllers, and for storage during job scheduling operations. The queue system is also used extensively by the record locking system.

INCREASING THE AVAILABLE QUEUE LIST SIZE

The number of queue blocks in use at any one time varies with system loading, number of users, and the kinds of tasks being performed. Some applications may demand a larger available list of queue blocks to insure safe system operation. A check is performed to see if the available queue is exhausted, and most operations wait for a queue block to become available. However, some critical operations cause a system halt if no queue blocks are available, because they cannot wait (for instance, an interrupt routine that, to free a queue block, must be dismissed before others can run). But, you can increase the size of the available queue list during system startup time.

The monitor is initially generated with enough queue blocks to boot the system and start processing the system command initialization file. Anywhere in the system initialization command file prior to the first SYSTEM command, you may execute the QUEUE nnn command which allocates ***nnn*** more queue blocks for general use. A typical increase for a large system with several users running extensive applications might be 5000 more blocks.

Once the system is up and running, you cannot add any more queue blocks to the list, so you must give your best estimate at your total requirements. The QUEUE command functions differently once the system is running. If you type the QUEUE command, the system responds by displaying the current number of free queue blocks in the available queue list. It is by this method, and by checking QFREE, that you may keep a close eye on the relationship between system operation and queue block usage.

QUEUE BLOCK USAGE BY THE SYSTEM

This section lists 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, this list will probably expand in future releases of the monitor. Also, add to this any programs you might have which include the QGET and QRET calls (described in the next section).

The SCSI dispatcher uses a large number of queue blocks to hold write-caching and SCSI command information.

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 I/O system uses the queue system to queue jobs waiting for access to a sharable device.

The record locking function of the record IO system uses queue blocks to keep track of what files are open, who they are open by, and what records are locked. One queue block is used for each job that has a file open. An additional queue block is used for each file that is open, regardless of how many jobs have it open. One queue block is used for each record that is locked within a file.

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

The XLOCK AlphaBASIC subroutine uses one queue block for each separate system lock 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.

QUEUE SYSTEM MONITOR CALLS

You can use the monitor queue system by using one of the four monitor queue management calls: QGET, QRET, QADD, and QINS. These calls are designed to be very fast so they can be used in interrupt level routines. The queue calls all perform an SVLOK to disable interrupts during their manipulation of the queue.

All calls require one argument which identifies the place to store the queue block address. Typically, this argument is an address register.

QGET - Obtain a Free Queue Block

This call obtains the first free queue block from the available list and returns its base address in the specified argument. It sets the Z-flag if the queue block was available, and resets it if no queue blocks were available. QGET removes the queue block from the available list, and then clears all words in the block to zeros. The call format is:

```
QGET     dst           ; dst gets address of queue block
BNE      error         ; check for no queue blocks avail.
```

QRET - Return a Queue Block

This call returns a queue block to the available queue list in the monitor. It returns the address which was in the first word of the block (usually a link to the next block in your chain) in the argument after it links the block back into the available queue list. The call format is:

```
MOV      address,src   ; get address of queue block to be returned
QRET     src           ; return the queue block
```

All queue blocks you have allocated by QGET, QADD, or QINS should eventually be returned to the monitor by the QRET call when they are no longer needed.

QADD, QINS - Manipulating Queue Blocks

Similar to the QGET call, these two calls obtain the first free queue block from the available list. They set the Z-flag if the queue block was available, and reset it if no queue blocks were available. If the queue block is available, they link it into your own specific list whose address is in the specified argument. 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 longword of the block. All other words in the queue block are available for the storage of specific data. The last block in the list contains a zero in longword 1 to mark the end of the list. The QADD call scans down the chain marked by the address in the argument, and then inserts the new queue block at the end of the existing list. The QINS call inserts the new queue block in the chain after the point indexed by the argument, and links the remaining list elements (if any) to the newly inserted block. Both calls then return the address of the second longword of the new queue block in the argument. This is the base of the data area of the queue block where you may store the data. The QADD call format is:

```
MOV      address,src/dst ; get address of start of queue
QADD     src/dst       ; add block to end of queue
BNE      error         ; error if no more queue blocks
                       ; src/dst now contains pointer to
                       ;   new queue block
```

The QINS call has the following format:

```
MOV     address,src/dst ; get address of place to insert new queue
                        ;   block
QINS    src/dst        ; insert a new queue block
BNE     error          ; error if no more queue blocks
                       ; src/dst now contains pointer to new queue
                       ;   block
```

Remember, the current size of each queue block is 36 bytes in length. The QADD and QINS calls place a link in the first four bytes, leaving 32 bytes 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.

QADDL, QINSL, QUNL - Link/Unlink Queue Block

These calls are similar to QADD and QINS, but differ in important ways. While QADD and QINS first allocate a queue block, and then link it into a specified queue, QADDL and QINSL assume the queue block has already been allocated by QGET, and handle linking only. For this reason, QADD and QINS return the address of long word 1 of the newly allocated queue block, assuming that the caller will then fill the queue block in; since QADDL and QINSL assume the block has already been allocated, and thus that the user has already filled the block in, QADDL and QINSL require the address of word zero of the block.

The call format is:

```
QADDL   queue,block
QINSL   queue,block
QUNL    queue,block
```

queue is the long word address of the queue. For the QADDL and QUNL calls, this is the base address; for QINSL, this is the insert position. The argument ***block*** is the longword address of longword 0 of the queue block to link; this address is allocated via QGET. Register or memory address format may be used for these arguments.

QADDL adds the specified block to the end of the specified queue. This operation is performed by scanning from the current position specified by the address in queue to its end and then linking in block.

QINSL inserts the block into the queue at the current position specified by the address in queue.

QUNL unlinks the block from queue by scanning from the current position looking for the address specified in block. QUNL does not deallocate the block.

These calls set the Z flag if the operation is successful, and clears it if the action is unsuccessful. The only unsuccessful operation recognized is an attempt to unlink a block from a queue to which it is not linked.

The current implementation of all of the queue management calls require that your program return to the pool all queue blocks allocated via QGET, QINS, and QADD before exiting. If this is not done, the queue block becomes unavailable until the computer is next rebooted.

Below is a sample program making use of the QADDL and QUNL calls. The program uses the stack to hold a queue block.

```
        SEARCH   SYS
        SEARCH   SYSSYM
MAIN:   LEA      A5,MAIN
        LEA      A4,BASE       ; Get base address of local queue.
        PUSH                   ; Provide place for queue block.
        QGET     @SP           ; Allocate a free queue block.
        QADDL    A4,@SP        ; Add the new block to the queue.
        QUNL     A4,@SP        ; Unlink the block just allocated
        QRET     @SP           ;   and return it to the free pool.
        POP                    ; Release the stack.
        BR       MAIN

BASE:   LWORD    0             ; Base address of local queue.
        END
```

For more information on the queue system, see the sections in this chapter on QGET, QRET, QINS, and QADD.

Chapter 6
The File Service System

AMOS has a simple yet powerful device-independent file service system which relieves the programmer of the task of IO coding for each device with which he wishes his program to interface. In addition to this device independence, AMOS contains all the necessary routines to manage the disk file system on a logical-call basis. The programmer does not need to be concerned with the exact physical placement of files on the disk except in rare instances when developing or testing system software. AMOS also contains an efficient means for developing new device drivers the programmer can incorporate into the system when it becomes necessary to interface with unsupported devices. This section gives a general overview of the file service system and describes the Dataset Driver Block (DDB) which is the descriptor link for all IO and file calls to the monitor.

AMOS supports two different file systems. The *traditional* file system is the one used by AMOS since the very first version. This traditional file system, however, imposes a limitation of 32 MB on the size of a single file. It also does not keep track of file protection or time and date stamps on an individual file basis. To support these features, AMOS also supports the *extended* file system. A single AMOS system can contain a mixture of both file systems, although doing so is intended primarily as a conversion aid rather than a permanent situation. This chapter assumes you will be using the extended file system, although it does point out where differences are visible to the programmer.

AMOS contains an integrated File/Record/Stream Locking Service which prevents two or more users from updating a file or record at the same time, or from encountering a deadlock situation when they each try to access a file or record the other has reserved. This service can lock entire files, single blocks, single records within files, or streams of bytes within USAM files. Locks are of two types: shared or exclusive. Shared locks permit multiple users to access a particular file or record, while exclusive locks allow only one user at a time to access that file or record. Record locks are used for random files shared with multiple users, while file locks are used to protect entire files of any type.

The various monitor calls that allow you to read and write files provide ways of letting you specify whether you want shared or exclusive access. They also allow you to specify whether you wish to wait for your request to be satisfied (i.e., until no other user has it locked), or if you wish control to be returned to your program immediately with a "resource in use" error.

By having a fully integrated locking service, AMOS makes it easy for you to generate multi-user software, without forcing you to implement your own multi-user protection.

THE DATASET DRIVER BLOCK

Monitor calls perform all IO operations and file operations by referring to a *DDB* which defines the device or file being operated upon. Whether the operation is performed on a unit-record device such as a printer, or on a specific file within a file structured device such as a disk, depends on the parameters passed to the monitor through the 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. There are no internal channel numbers or device numbers to limit the number of concurrently active devices or files. We can sum up the general sequence of events for the complete processing of a device or file operation like this:

1. The DDB contains 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.M68[101,1], by an FSPEC call.
2. The IO 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, normally by 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, normally by a CLOSE call referencing the DDB.

AMOS contains complete error processing routines which allow you to specify (by flags in word 1 of the DDB) whether any uncorrectable errors will cause an automatic error message to the operator on his terminal, an aborting of the program and return to monitor, or both. You may also elect to process the errors yourself by checking the error code returned in word 1 of the DDB.

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 IO operations can take place. The size of a DDB is defined by the symbol D.DDB. You can assign any label you wish to the DDB. This label then becomes the reference label for all subsequent operations to that dataset. You must set up some of the items in the DDB before certain operations may be called for, while other items are set up and used by the monitor file service routines.

We have assigned each of these entries a symbolic offset of the form D.xxx, which is defined in SYS.UNV. You should use these symbolic offsets at all times, as the actual positions of the various fields within the DDB are subject to change. The following descriptions explain the use of each item.

0	Flags	Error code	D.FLG / D.ERR
2	Device name		D.DEV
4	Device unit number		D.DRV
6	Filename		D.FIL
10			
12	Extension		D.EXT
14	Project/programmer number		D.PPN
16	Block number		D.REC
20			
22	Buffer address		D.BUF
24			
26	Buffer size		D.SIZ
30			
32	Buffer index		D.IDX
34			
36	Call level	Open code	D.LVL / D.OPN
40	User argument		D.ARG
42			
44	Device driver address		D.DVR
46			
50	CPU specification		D.CPU
52			
54	Device format flags		D.FMT
56	Directory marker		D.DIR
60			
62			
64			
66	Auxiliary storage		D.AUX
70			
72	Creation date/time		D.CDT
74			
76	Update date/time		D.UDT
100			

Figure 6-1: Dataset Driver Block (DDB) Format

Offset	Field	Symbol
102 104	Backup date/time	D.BDT
106	Record size	D.RSZ
110 112	Number of blocks in file	D.FSZ and D.WRK
114 116	Bytes in last block	D.LSZ
120 122	Block number of first block in file	D.BAS
124 126	Type Code	D.TYP
130 132	Protection	D.PRT
134 136	File ID	D.FID
140 142	Reserved	
144 146	Reserved	

Figure 6-1: DDB Format (continued)

Error Code (D.ERR)

This byte is set to a non-zero code at the completion of an IO operation that was unsuccessful for various reasons. A zero indicates the operation was successful. You need to test this byte only if the error control flag in the flags byte (DDB+D.ERR) 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.

Flags (D.FLG)

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

Symbol	Octal Value	Hex Value	Meaning
D$ERC	1	1	Set by user to force a return on error condition (abort to the monitor if this flag is zero).
D$BYP	2	2	Set by user to bypass printing of error messages on error conditions
D$UPD	4	4	File has been updated (internal file service use only)
D$RFD	10	8	This file has been "translated" by the VDK software
D$XFI	20	10	Transfer initiated (internal file service use only)
D$RWF	40	20	Read if zero or write if one (used by device drivers)
D$INI	100	40	Device is INITed—set by the INIT call or by the user if an explicitly defined buffer is in use.
D$DSB	200	80	DDB is busy (transfer has been initiated or queued)

Device (D.DEV)

Before any IO operations may be performed, either an FSPEC call or your program itself must set the 3-character device code (packed RAD50). A zero in this field causes the monitor to use the default (login) device.

Drive (D.DRV)

This word specifies the drive to be used for the transfer (only drivers for devices with multiple drives use this word). If the device is DSK, the default drive is the one you are currently logged into. Other devices may have different defaults. You may use a -1 (octal 177777) to indicate the current default drive number.

Filename and Extension (D.FIL and D.EXT)

These three words contain the RAD50 packed filename and extension for file structured devices. Drivers for devices which are not file structured ignore these words. However, if an error occurs with a non-file-structured device, you may receive inaccurate error messages if these three words are not set to zero values.

PPN (D.PPN)

The next word contains the octal project-programmer number for the area to be used to locate the file. It is used only on file structured devices which are multi-user based (such as disks). A zero causes the default value to be the current PPN the job is logged in under. To prevent inaccurate error messages, this word should be zero if not used.

Block Number (D.REC)

You must set this 32-bit block number to read or write a specific random block from a random access device such as disk. When performing physical IO by using the READ and WRITE calls, this block number is the absolute block number expressed as an offset from the start of the device. The first block on the device is considered block zero, and the block numbers increment sequentially from there.

When performing logical record IO on a file open for random processing (via OPENR), this field contains the relative block number within the file. Each block within a random file is 512 bytes long, with the first block being block number zero, with subsequent block numbers being numbered sequentially.

When performing logical record IO on a file open for record processing (via OPENIO), this field contains the relative *record* number within the file. The record size is defined on a file basis (the size of the record being stored in D.RSZ), with the first record being record number zero.

Most non-disk devices are not random access, in which case the respective drivers ignore this block number.

Buffer Address (D.BUF)

This is the 32-bit absolute address of the base of the buffer to use 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 D$INI flag in DDB+D.FLG which indicates a buffer has been allocated either by the INIT call or directly by your program. No transfers can take place without a buffer.

Buffer Size (D.SIZ)

This is the size in bytes used for the physical transfer operation. The READ call transfers this number of bytes from the device to the user buffer beginning with the address in DDB+D.BUF. The WRITE call transfers this number of bytes from the user buffer to the user device. The INIT call sets this size to the standard buffer size, or you can set the size yourself if you are doing your own buffering. You may modify the size if you need to transfer blocks of variable sizes. 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.

Buffer Index (D.IDX)

This is a 32-bit counter which is used by logical routines (INPUT and OUTPUT calls) for keeping track of bytes transferred into and out of the buffer. Various calls reset this value, and you can then use it and increment it as bytes are transferred into and out of the buffer. Later sections describing these calls give more complete details. This buffer index word is normally not a true buffer pointer, but rather an offset from the buffer base (per DDB+D.BUF) to the current byte being manipulated.

Open Code (D.OPN)

The OPEN call sets this byte 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 you should not modify it. The following open codes are in use:

Symbol	Octal Value	Hex Value	Meaning
D$OPNN	0	0	File is not open
D$OPNI	1	1	File is open for sequential input (OPENI call)
D$OPNO	2	2	File is open for sequential output (OPENO call)
D$OPNR	4	4	File is open for random input/output (OPENR call)
D$OPNA	10	8	File is open for appending (OPENA call)
D$OPNS	22	12	File is open for superseding (OPENS call)
D$OPNC	44	24	File is open for record IO (OPENIO call)

Call Level (D.LVL)

This byte is for internal use only; it 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.

User Argument (D.ARG)

Some of the logical file calls use this 32-bit field to pass an argument to the call. For example, the DSKCTG call, which allocates a contiguous data file, expects the size of the desired file to be passed in D.ARG. You can find details on the requirements for the argument in this field under the individual calls in Section 6.2.

If LOKSER rejects an I/O request (for example, if you try to write a record someone else has locked), the JCB of the job holding the record lock is returned in D.ARG of the DDB.

Device Driver Address (D.DVR)

This longword contains the address of the device driver being used to access the device.

If a DDB is to be used for handling different files, this field must be cleared each time the DDB is set up to refer to a different device. Failure to do so will result in data corruption. To avoid this problem, the best course is to clear the entire DDB to zero each time it is to be re-used.

CPU Specification (D.CPU)

When accessing devices on different CPUs (via AlphaNET), this longword will contain the CPU ID of the system being accessed. It must be set to zero when accessing devices on the local system.

Device Format (D.FMT)

This 16-bit field is used by the AMOS file system to keep track of the format of the directory being accessed. It is normally used by user programs only to check to see if the file being accessed is in a traditional or extended format directory. The field will contain a zero for traditional format directories and a 2 for extended format.

Directory Marker (D.DIR)

This field is made up of two 16-bit fields and one 32-bit field. It is used by the AMOS file system to keep track of the current position within a directory being accessed. This field is for internal use and it is not expected it will be accessed by user programs.

Auxiliary Storage (D.AUX)

This field is reserved by the AMOS file system for internal use.

Creation Date and Time (D.CDT)

This 32-bit field contains the date and time the file was created, in a packed format. This packed format may be unpacked with the $UNPDT subroutine, discussed in Appendix D.

This field is not used by the traditional file system and will be zero when that file system is in use.

Update Date and Time (D.UDT)

This 32-bit field contains the date and time the file was last modified, in a packed format. This packed format may be unpacked with the $UNPDT subroutine, discussed in Appendix D.

Sequential files are considered modified if they have been opened and written to in append mode. Random files are considered to be modified any time they have been opened and written to. This field is updated at the time the file is closed.

This field is not used by the traditional file system and will be zero when that file system is in use.

Backup Date and Time (D.BDT)

This 32-bit field contains the date and time the file was last backed up, in a packed format. This packed format may be unpacked with the $UNPDT subroutine, discussed in Appendix D.

The backup date and time are set by the various backup utility programs. This field is normally used to determine if a file has been modified since the last time it was backed up, thereby determining whether or not the file needs to be backed up again.

This field is not used by the traditional file system and will be zero when that file system is in use.

Record Size (D.RSZ)

This 16-bit field contains the default record size associated with this random file. This field will be valid after a LOOKUP call, and is used by the DSKCTG call to select the default record size.

The default record size is used when opening a file for record IO via the OPENIO call. While you must always specify the record size when opening a file for record IO, this field provides a method of checking to make sure the correct record size is being provided.

If this field contains a zero, no default record size has been set for this file, and no check for record size validity will be made.

This field is not used by the traditional file system and will be zero when that file system is in use.

File Size (D.FSZ)

This 32-bit field contains the number of disk blocks occupied by the file. This field is valid after a LOOKUP or OPENx call.

Last Block Size (D.LSZ)

This 32-bit field contains the number of bytes contained in the last block of a sequential (linked) file. If the file is a random (contiguous) file, this field will have a -1 placed in the low-order word. This field is valid after a LOOKUP or OPENx call.

Base Block Number (D.BAS)

This 32-bit field contains the block number of the first disk block occupied by this file. This field is valid after a LOOKUP or OPENx call.

Type Code (D.TYP)

This 32-bit field is used to store special attributes about the file when the extended file system is in use. This field is valid after a LOOKUP or OPENx call. This field is also used by the directory access monitor calls described in Chapter 18.

The high-order word of this field contains a series of flags used to describe the attributes of this file. The flags are:

Octal Value	Hex Value	Meaning
0	0	End of directory
1	1	Current item is a system entry
2	2	Current item is a directory entry
4	4	Current item is a data file
20	10	This is a USAM file that has been initialized
100000	8000	Current item has been deleted

The low-order word of this field contains the size of the filename for this entry. If the entire longword field is zero, this item signifies the end of the directory. This field is not used by the traditional file system.

Protection (D.PRT)

This 32-bit field is used to store the protection level associated with this file when the extended directory structure is in use. This field is valid after a LOOKUP or OPENx call.

The protection field is broken into five six-bit fields. Each six-bit field specifies the protection level which applies to a given class of user attempting to access the file. On each file access, AMOS determines the class of user making the access, accesses the proper six-bit field, and uses that value to determine whether access is allowed. The five groups (and their associated bit fields) are defined as follows:

Group Number	Bit Numbers	Meaning
1	0 - 5	Users within the same directory (PPN)
2	6 - 11	Users at the same directory level (same project number)
3	12 - 17	Users at a different directory level (different project number)
4	18 - 23	Users within the same network group
5	24 - 29	All other users not included in groups 1-4
	30 - 31	Reserved

Each six-bit field is encoded as a binary number, with each bit having a distinct characteristic. The bits are defined as follows:

Bit Number	Meaning
0	File may be read
1	File may be written
2	File may be executed (FETCHed)
3	File may be deleted, renamed, or have its protection changed
4	Reserved for future use
5	Reserved for future use

Certain combinations of these bits imply various capabilities. If a file does not have the READ bit set, it is "invisible," that is, it does not appear in directory listings or via the directory access calls.

Protection bit 3 (delete, rename) is overridden by logging in to OPR:, allowing the system operator to delete (or re-protect) files that otherwise would be inaccessible.

Bit 30 of D.PRT is used to flag the file should be zeroed before it is deleted. For additional data security, you may wish to have all blocks of a file overwritten with zeros prior to the file being deleted and having its data blocks returned to the pool of free blocks.

Bit 31 of D.PRT is reserved for future use.

When a file is created (via OPENO, OPENS, or DSKCTG), it is given the protection specified by the JOBDFP field of the creating job's JCB, allowing each job to have a different default protection.

The D.PRT field is always displayed as either an octal number (never hex or decimal), or as a descriptive textual listing of capabilities. This field is not used by the traditional file system.

File Locking ID (D.FID)

This longword field is used by the file and record locking system to store a unique file ID assigned to the file while it is open. This field is not used by non-file structured devices.

Driver Work Area (D.WRK)

The seven longwords starting at D.FSZ are considered to be the driver work area. This area is available for use by non-file structured device drivers as impure storage space during execution of the driver.

Device Transfer Buffers

Each DDB must have an associated transfer buffer to handle input and output operations. 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 allocates 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—as long as it is large enough for any logical calls to be performed—and then set its address in DDB+D.BUF. See Section 6.2 which discusses the IO calls themselves for more details on using these buffers.

Error Handling

When an error occurs during any file service call, the file service routines 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 the D$ERC flag and the D$BYP flag in the flags byte at DDB+D.FLG.

1. D$BYP is tested and if it is not set, the monitor outputs a standard error message to the user terminal, giving the type of call that failed, the file specification for the device the error occurred on, and the reason for the error. The appropriate error code is also placed in the error byte at DDB+D.ERR for later testing by the user.
2. D$ERC of the flags byte is tested; if it is not set, the user program is aborted by the file service system, and you are returned to monitor mode. You normally set these flags on before any IO calls are made, if you wish to process the errors within the user program itself.

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, and the symbol assigned to the particular error number (in SYS.UNV):

Symbol	Octal Value	Hex Value	Meaning
D$ESPC	1	1	File specification error (FSPEC)
D$EMEM	2	2	Insufficient free memory for buffer allocation (INIT)
D$EFNF	3	3	File not found (OPENI, OPENR, OPENA, DSKDEL, DSKREN)
D$EFAX	4	4	File already exists (OPENO)
D$ERDY	5	5	Device not ready (all calls)
D$EFUL	6	6	Device full (OUTPUT, DSKCTG, CLOSE, DSKALC)
D$EERR	7	7	Device error (all calls)
D$EUSE	10	8	Device in use (ASSIGN)
D$EILC	11	9	PPN does not exist (all file calls)
D$EPRV	12	A	Protection violation (OPENO, OPENR, DSKDEL, DSKREN)
D$EWRT	13	B	Write protected (all output calls)
D$ETYP	14	C	File type mismatch
D$EDNX	15	D	Device does not exist (all calls)
D$EIBN	16	E	Illegal block number (READ, WRITE)
D$EINI	17	F	Buffer not INITed (all calls except INIT)
D$EFNO	20	10	File not open (READ, WRITE, INPUT, OUTPUT, CLOSE)
D$EFAO	21	11	File already open (all OPEN calls)
D$EKPT	22	12	Bitmap kaput (all disk bitmap calls)
D$EMNT	23	13	Device not mounted (all calls)
D$EIFL	24	14	Invalid filename (OPENO, FSPEC, DSKCTG)
D$EBBH	25	15	BADBLK.SYS has a bad hash total (DSKMNT)
D$EBBW	26	16	BADBLK.SYS is in wrong (unsupported) format (DSKMNT)
D$EBBN	27	17	BADBLK.SYS not found (DSKMNT)
D$ENOQ	30	18	Insufficient queue blocks (all calls)
D$EMFD	31	19	MFD is damaged (all file calls)
D$ELNM	32	1A	First logical unit is not mounted
D$ERNR	33	1B	Remote is not responding (reserved for future use)
D$EFIU	34	1C	File in use
D$ERIU	35	1D	Record (or stream) in use
D$EEMB	36	1E	Deadly embrace possible
D$EDEL	37	1F	File cannot be deleted (obsolete)
D$EREN	40	20	File cannot be renamed (obsolete)
D$ERNL	41	21	Record (or stream) not locked
D$ERNO	42	22	Record not locked for output
D$ELQF	43	23	LOKSER queue is full
D$ENFS	44	24	Device is not file structured
D$EIRS	45	25	Illegal record size
D$EBLK	46	26	Block allocate/de-allocate error
D$EIAA	47	27	Invalid argument address
D$EARG	50	28	Invalid argument
D$ENBA	51	29	No blocks allocated, cannot open for append
D$EUFD	52	2A	UFD is damaged
D$EFSZ	53	2B	Illegal file name size in DDB (reserved)

You may convert a DDB error code to its text message through the use of the ERRMSG monitor call (see Chapter 13).

At the conclusion of every file service monitor call, the monitor tests the error byte at the base of the DDB for the convenience of your 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, only applies if you have the error trapping bit set in the DDB status word to prevent the job from being aborted on a file error.

FILE SERVICE MONITOR CALLS

This section describes the file service calls which are available to your programs for both logical and physical IO 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 section for a complete description of the DDB format. In brief, the calls described in this section are:

Call	Purpose
FSPEC	Process a device specification
INIT	Initialize a dataset driver block buffer
LOOKUP	Look up a file to see if it exists
OPENI	Open a file for sequential input
OPENO	Open a file for sequential output
OPENA	Open a file for appending
OPENS	Open a file for sequential output, superseding current file
OPENR	Open a file for random input/output
OPENIO	Open a random file for record IO
CLOSE	Close a file to further processing
CLOSEK	Close a file but keep it locked
READ	Read a physical block
WRITE	Write a physical block
INPUT	Read a logical block
INPUTL	Read a logical block with locking
INPUTX	Read a logical block without regard to locking
OUTPUT	Write a logical block
OUTPTL	Write a logical block with locking
FILINx	Input a byte, word, or longword
FILOTx	Output a byte, word, or longword
LOCKF	Lock a file
UNLOKF	Unlock a file
UNLOKR	Unlock a record
DSKDEL	Delete a file
DSKREN	Rename a file
CHPROT	Change a file's protection
DSKCTG	Create a contiguous file
ASSIGN	Assign a device to a job
DEASGN	De-assign a device from a job
DSKMNT	Mount a device
DSKUMT	Unmount a device

FSPEC - Process an ASCII Filespec

FSPEC processes an ASCII file specification from a command line (or any other ASCII buffer) and sets up the parameters in the DDB according to the results of the processing. The file specification must be indexed by A2 and be in the standard format of:

```
        cpu-devn:{devn:}filnam.ext[p,pn]
```
or:
```
        cpu-devn:{devn:}[p,pn]filnam.ext
```

with a valid termination character, if you use a short default specification.

More than one device specification is allowed so you can specify ersatz names or network device names in the filespec. If more than one device specification is present, the "last" or rightmost is the one that will be present in the DDB.

The FSPEC call is slightly different from the rest of the IO calls in that it allows you to use a second argument if you wish. This argument must be the default extension for the filename parameter to be used if 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 uses the contents of D6 as the default file extension. If D6 contains a -1, the FSPEC processor stops scanning as soon as it has found a device specification, ignoring any following filename. The general calling sequence is:

```
        FSPEC   adr{,ext}               ; load DDB at adr with filespec
```

FSPEC recognizes ersatz devices by scanning the filespec for device specifications that could be ersatz devices. If it finds some, it checks the names against the ersatz device table—if it does not find a match there, it assumes it to be a device name, and processes it accordingly.

If the file specification contains a device code (marked by a terminating colon), the first three characters are packed RAD50 and stored in DDB+D.DEV. The drive number, if specified, is stored in DDB+D.DRV. If the device code is not specified, a zero is stored in DDB+D.DRV. If the drive number is not in the input specification, an octal 177777 is stored in DDB+D.DRV to flag the default drive number to the device driver. The operating system treats these default values as the current logged in device (stored in the job's JCB items JOBDEV and JOBDRV).

FSPEC then processes the filename and extension. If the call contained no default extension as its second argument, the FSPEC processor returns to your program at this point. Otherwise, the filename and extension are packed RAD50 and stored in the three words at DDB+D.FIL and DDB+D.EXT. If no filename is entered in the input specification, the word at DDB+D.FIL is cleared to zero to flag the absence of the filename parameter. If a filename is entered without an extension, the default extension specified in the second argument of the FSPEC call is stored as the extension in DDB+D.EXT.

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

At the conclusion of the processing of the input file specification, the index A2 is pointing to the termination character (the first character following the file specification string). If an error in the input string is detected, the monitor displays the "?File specification error" message (unless suppressed by the D$BYP flag in DDB+D.FLG), and the program is aborted (unless suppressed by the D$ERC flag in DDB+D.FLG). The error code 01 is also set in the DDB+D.ERR error code byte.

No other modifications to the DDB area take place, except the error byte at DDB+D.ERR is cleared at the start of the FSPEC processing. If you do not use the FSPEC call to set up your DDB, you must use some other form of explicit code to insure the DDB is set up properly to define the device and file for any subsequent IO operations.

The PPN specification allows "errors" in the form of substituting a "." or "m" for the comma. This is compatible with the LOG program.

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 locates the device driver (searching [1,6] on DSK0: if not in memory), then allocates a standard size buffer based on the buffer size specified in the driver. The D$INI flag in DDB+D.FLG is set to indicate initialization has been performed. INIT sets the address of the buffer into DDB+D.BUF, and sets the size in bytes into DDB+D.SIZ. If you do not use the INIT call and perform the buffer set-up yourself, DDB+D.DVR ***must*** be set to zero before you perform any file calls. Setting DDB+D.DVR to zero forces the device driver to be located (and if necessary, loaded) by the next file call. The calling sequence for INIT is:

```
INIT    adr          ; initialize DDB at adr
```

No calls de-allocate 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 is de-allocated automatically when the program exits to the monitor, or it can be explicitly de-allocated by using the DELMEM call with the address stored in DDB+D.BUF. Recall that the buffer is allocated as a standard memory module with a GETMEM call.

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 or equivalent code for processing.

If the D$INI flag in DDB+D.FLG is already set when an INIT call is performed, INIT ignores the flag and allocates another buffer.

Find the File

This is a form of the OPEN calls 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; you must use an OPENI call if you decide to read from the file later on. The LOOKUP call is useful for determining if a file you want to create already exists, so you can delete it first by the DSKDEL call. The LOOKUP call is ignored for devices which are not file structured.

LOOKUP returns any of the standard error codes (e.g., protection violation, file not found, etc.). However, note the byte returned is ***negative***. Before comparing it to the standard error codes, remember to negate the number first. The calling sequence for LOOKUP is:

```
LOOKUP  adr          ; see if file exists
BNE     not found    ; branch if not found
```

The LOOKUP call is also useful for some system programming techniques, since it returns parameters about the file in the DDB work area. If LOOKUP finds the file, it loads the last three words of the user file directory (UFD) item into the three longwords of the DDB work area. These three longwords are the number of blocks in the file, the number of active data bytes in the last block, and the block number of the first data block in the file. See Appendix A for complete details on the directory format.

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 OPENI cannot find the file. It places the code D$OPNI into DDB+D.OPN to flag the OPENI operation. The OPENI call is normally followed by a series of INPUT calls which deliver sequential blocks from the device to the file buffer. The calling sequence is:

```
OPENI    adr{,flags}   ; open file for sequential input
```

adr references a DDB specifying the file to be opened, and ***flags*** is an optional argument which can contain the following bits:

Symbol	Meaning
F.WAT	User will wait for access to the file
F.EXC	Request exclusive use of file
F.NEX	Bypass nolock options

The OPENI call defaults to shared access to the file, with no waiting, if no flags are specified. If a file has already been opened with the OPENA or OPENO monitor calls, and any OPEN call is used to open it again without closing it first, you get a D$EFIU (?File in use) error. (See the *AMOS File Locking Manual* for information on the different file locking modes.)

The OPENI call is ignored for devices which are not file structured.

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 the file does not exist, OPENO sets up the DDB for OUTPUT processing. It places the code D$OPNO into DDB+D.OPN to flag the OPENO operation. The OPENO call is normally followed by a series of OUTPUT or FILOTx calls which transfer data from the buffer to sequential blocks in the file. The calling sequence is:

```
OPENO    adr{,flags}   ; open file for sequential output
```

adr references a DDB specifying the file to be opened, and ***flags*** is an optional argument which can contain the following bits:

Symbol	Meaning
F.WAT	User will wait for access to the file
F.NEX	Bypass nolock options

The OPENO call defaults to exclusive access to the file, without waiting, if no flags are specified. If a file has already been opened with the OPENA or OPENO monitor calls, and any OPEN call is used to open it again without closing it first, you get a D$EFIU (?File in use) error. (See the *AMOS File Locking Manual* for information on the different file locking modes.)

The OPENO call is ignored for devices which are not file structured.

OPENS - Open a File for Output, Superseding Any Existing File

The OPENS call first searches the specified device in the specified user area and deletes any already existing file with the same name. OPENS then sets up the DDB for OUTPUT processing, just as with the OPENO call, and places the code D$OPNS into DDB+D.OPN to flag the OPENS operation. The calling sequence is:

```
              OPENS   adr{,flags}   ; open file for sequential output
```

adr references a DDB specifying the file to be opened, and ***flags*** is an optional argument which can contain the following bits:

Symbol	Meaning
F.WAT	User will wait for access to the file
F.NEX	Bypass nolock options

The OPENS call defaults to exclusive access to the file, without waiting, if no flags are specified. If a file has already been opened with the OPENA or OPENO monitor calls, and any OPEN call is used to open it again without closing it first, you get a D$EFIU (?File in use) error. (See the *AMOS File Locking Manual* for information on the different file locking modes.)

The OPENS call is ignored for devices which are not file structured.

OPENA - Open and Append to Existing File

The OPENA call is similar to OPENO, except it allows you to append data to an existing file. It places the code D$OPNA into DDB+D.OPN to flag the OPENA operation. The OPENA call is normally followed by a series of OUTPUT calls which transfer data from the buffer to the end of the existing file. The calling sequence is:

```
              OPENA   adr{,flags}   ; open file for sequential append
```

adr references a DDB specifying the file to be opened, and ***flags*** is an optional argument which can contain the following bits:

Symbol	Meaning
F.WAT	User will wait for access to the file
F.NEX	Bypass nolock options

The OPENA call defaults to exclusive access to the file, with no waiting, if no flags are specified. If a file has already been opened with the OPENA or OPENO monitor calls, and any OPEN call is used to open it again without closing it first, you get a D$EFIU (?File in use) error. (See the *AMOS File Locking Manual* for information on the different file locking modes.)

This call is ignored for devices which are not file structured.

OPENR - Open a File for Random Processing

The OPENR executes basically the same as the OPENI call, but the code stored in DDB+D.OPN is D$OPNR 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 blocks in the file and the buffer in both directions. The calling sequence for OPENR is:

```
              OPENR    adr{,flags}   ; open file for random processing
```

adr references a DDB specifying the file to be opened, and ***flags*** is an optional argument which can contain the following bits:

Symbol	Meaning
F.WAT	User will wait for access to the file
F.EXC	Request exclusive use of file
F.RON	Read-only mode
F.NEX	Bypass nolock options

The OPENR call defaults to shared access to the file, with no waiting, if no flags are specified. The file being used in read only mode can also be opened by another user for exclusive access. When exclusive access is granted, the read only user is marked so that the next time the read only program tries to access the file, the D$EFIU "?File in use" error is returned in D.ERR of the DDB. Read only mode does not allow creating or updating records; INPUT, INPUTL, INPUTX, and UNLOKR operations are allowed. (See the *AMOS File Locking Manual* for information on the different file locking modes.)

The OPENR call is ignored for devices which are not file structured.

OPENIO - Open a File for Record IO

The OPENIO opens a file for record IO processing, placing the code D$OPNC in DDB+D.OPN to flag the operation. Rather than doing your own blocking and deblocking, as you must with a file opened via OPENR, a file opened for record IO allows AMOS to perform all blocking functions for you. In addition, because AMOS is dealing with *records* not *blocks*, it is able to perform multi-user locking at the record level.

Files opened for record IO must be contiguous files, just as with OPENR. After opening a file via OPENIO, you will perform PUT and GET calls (or PUTL and GETL calls) to perform the actual IO. The calling sequence for OPENIO is:

```
      OPENIO  adr,flags,size        ; open file for record processing
```

adr references a DDB specifying the file to be opened, ***flags*** contains flag bits described below, and ***size*** specifies the logical record size to be used.

Symbol	Meaning
F.WAT	User will wait for access to the file
F.EXC	Request exclusive use of file
F.BIG	Span records across blocks and allow records larger than 512 bytes.
F.RON	Read-only mode
F.NEX	Bypass nolock options

If the file being opened is within an extended directory structure, and a logical record size was specified when the file was created, the OPENIO call validates the record size specified in the OPENIO call and returns an error (D$EIRS) if it does not match. If the file being opened is within a traditional directory structure, or if no record size was specified when the file was created, no validation of record size occurs, and the record IO system simply uses the value you supply in the OPENIO call.

The AMOS record IO system supports two methods of calculating logical records. The first, which is compatible with AlphaBASIC, limits all records to 512 bytes or less, and aligns the records so they never span a block boundary. The other method, which is compatible with AlphaCOBOL, allows records larger than 512 bytes (up to 65535 bytes) and does not pay attention to block boundaries, spanning them as necessary. The OPENIO call defaults to the smaller, AlphaBASIC compatible mode. If you wish to use the other mode, you must specify the F.BIG flag in the OPENIO call, and with all subsequent IO operations.

The OPENIO call defaults to shared access to the file, with no waiting, if no flags are specified.

The file being used in read only mode can also be opened by another user for exclusive access. When exclusive access is granted, the read only user is marked so that the next time the read only program tries to access the file, the D$EFIU "?File in use" error is returned in D.ERR of the DDB. Read only mode does not allow creating or updating records; INPUT, INPUTL, INPUTX, and UNLOKR operations are allowed. (See the *AMOS File Locking Manual* for information on the different file locking modes.)

The OPENIO call is ignored for devices which are not file structured.

CLOSE - Close a File

The CLOSE call finishes up logical processing of a file and clears the open code in DDB+D.OPN. No further INPUT or OUTPUT operation may occur once a file has been closed. No action is normally taken on a file which is open for input. For files open for output, the final block is written out and the file is added to the directory system on the specific device. For files open for appending, the final block is written out and the directory information is updated. The calling sequence is:

```
        CLOSE   adr         ; close the file
```

The CLOSE call is ignored for devices which are not file structured.

CLOSEK - Close a File and Keep Locked

CLOSEK is identical to CLOSE except that, unlike CLOSE, it does not unlock the file being closed. The CLOSEK call finishes up logical processing of a file and clears the open code in DDB+D.OPN. No further INPUT or OUTPUT operation may occur once a file has been closed. No action is normally taken on a file which is open for input. For files open for output, the final block is written out and the file is added to the directory system on the specific device. For files open for appending, the final block is written out and the directory information is updated. The calling sequence is:

```
              CLOSEK   adr           ; close the file
```

CLOSEK closes the specified file, but also causes it to remain locked until the job that opened the file re-opens the file or explicitly unlocks it, or until LOKUTL is used to unlock it. Using CLOSEK causes the file to be closed in FILSER, but inside LOKSER, the file and its locked records will remain locked (with a "keep" flag set), even after the program exits. When the file is opened again later by the same job that originally opened it, the "keep" flag will be reset, and the locked records, if any, will be released. The locking status (exclusive, shared, or read only) of the file will be as defined by the new open call.

If the file was opened for exclusive use by the open call before CLOSEK, the only job that can access it after CLOSEK is the job that first opened it. If the file was opened for shared use, other jobs can access it for shared use after the CLOSEK but, again, only the job that originally opened it can unlock it so it can be used in exclusive mode.

A file can remain permanently locked if the program that opened it aborts after the CLOSEK and before re-opening it. In this case, the System Operator must use the LOKUTL utility to unlock the file.

If a file was opened for shared use and closed by CLOSEK, the job that opened it will not be able to open it again for exclusive use if some other job has opened it for shared use. But, in this situation, the original job will have no problem re-opening the file for shared use. See the *AMOS File Locking Manual* for information on the different file locking modes.

The CLOSEK call is ignored for devices which are not file structured.

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 READ is not a logical file call. The calling sequence is:

```
              READ     adr           ; read block from device
```

Sequential Access Devices

For sequential access devices such as a paper tape reader, the READ call delivers one block from the device to the user buffer. The size of this block is normally the number of bytes specified in DDB+D.SIZ, 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+D.SIZ (such as a tape reader which runs out of tape), a lesser number may be transferred, in which case the count in DDB+D.SIZ is adjusted to reflect the true number actually transferred to the user buffer.

Random Access Devices

For random access devices such as a disk, you must specify the block number to be located and read by placing that number into DDB+D.REC before executing the READ call. Most random access devices always transfer the requested number of bytes per DDB+D.SIZ into the user buffer. (If the buffer is larger than the physical block, the system reads multiple contiguous blocks to fill up the buffer.) An error results if the block number is not within the range of the specific device. For example, the standard AMOS floppy disk is structured as 500 (decimal) blocks of 512 bytes each. The legal block numbers, therefore, range from 0 through 499, decimal. Similar range restrictions apply for each random device.

Interrupt Structure

The system allows certain interrupt driven devices to overlap their IO with the processing of other jobs. Normally, the execution of a READ (or WRITE) call results in the driver waiting for execution to be complete before returning to the user or allowing other users to run. With some devices, usually disks, the execution of a READ call suspends the running of the user program until the transfer has been completed, at which time the user job re-activates. During the time your job is suspended, other jobs are allowed to run, increasing the total system throughput.

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. The calling sequence for WRITE is:

```
        WRITE   adr          ; write a block to the device
```

Sequential Devices

For sequential access devices, such as a printer, the WRITE call delivers one block to the device from the user buffer. The size of this block is the number of bytes specified in DDB+D.SIZ. The driver is responsible for the correct transfer count, and you may alter the number in DDB+D.SIZ for each new WRITE call to the same device for the writing of variable length records.

Random Devices

For random access devices, such as disks, you must specify the block number to be located and written, by placing that number into DDB+D.REC before executing the WRITE call. Most random access devices always transfer the requested number of bytes per DDB+D.SIZ from the user buffer. An error results if the block number is not within the range of the specific device.

Interrupt Structure

The interrupt structure of the WRITE call functions the same as that of the READ call discussed above.

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 calling sequence is:

```
              INPUT    adr{,flags}   ; get block from input device
```

The INPUT call first sets the standard buffer size into DDB+D.SIZ, so you may not use this call to transfer non-standard record sizes. The number of bytes actually read may be less than the standard block size due to the driver processing or due to an end-of-file condition. The actual number of bytes transferred is set into DDB+D.SIZ by the driver routine.

Sequential File Processing

The INPUT call is mainly used in logical sequential file processing; it sets up the buffer index value in DDB+D.IDX 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+D.IDX. For unit record devices, the value is zero since all data within the buffer is user data. For sequential disk files, however, the first part of each block within the file is a link to the next block. Therefore, the disk driver sets the value in DDB+D.IDX so processing starts with the byte after the link in the user buffer.

The optional ***flags*** argument is not used for sequential file processing.

Since you will usually process sequential input files to read a byte, word, or longword, you will almost always access these files using the FILINx calls (which use INPUT), rather than by direct use of the INPUT call. (See the Section "FILINB, FILINL, FILINW - Input from a Device," below.)

Random File Processing

A special situation arises for files opened for random access by the OPENR call. Instead of reading the next sequential block, INPUT reads the specific relative block whose number is in DDB+D.REC into the user buffer.

You first set this number up and then execute the INPUT call. The block number is actually relative to the base of the file and has no direct relationship to the physical block on the device as would be returned by a READ call.

The INPUT call reads the requested file block but does not leave it locked, making the INPUT call the one to use when reading data that will not be updated. If you wish to update and rewrite the block you are reading, use the INPUTL call, which locks the block for exclusive use, allowing subsequent updates.

The ***flags*** argument allows you to specify whether or not you wish your program to wait for access to the block, which may be locked by another user. This optional argument can contain the following flag bit:

Symbol	Meaning
F.WAT	User will wait for access to the block

The INPUT call defaults to not waiting if no flags are specified.

Special Devices

For devices that do not implement special processing of logical calls, the INPUT call performs a READ call instead.

INPUTL - Perform a Logical Read with Locking

The INPUTL call is identical to the INPUT call, except that when used with files open for random access (OPENR), it leaves the requested block locked for exclusive use, allowing for subsequent update. The calling sequence is:

```
INPUTL   adr{,flags}   ; get block from input device
```

Sequential File Processing

For sequential files, the INPUTL call is identical to the INPUT call.

Random File Processing

Like the INPUT call, INPUTL reads the specific relative block whose number is in DDB+D.REC into the user buffer.

You first set this number up and then execute the INPUTL call. The block number is actually relative to the base of the file and has no direct relationship to the physical block on the device as would be returned by a READ call.

The INPUTL call reads the requested file block and leaves it locked for exclusive use, allowing you to rewrite the block (via OUTPUT) or unlock it (via UNLOKR). If you do not wish to update and rewrite the block you are reading, use the INPUT call, which does not leave the block locked.

The ***flags*** argument allows you to specify whether or not you wish your program to wait for access to the block, which may be locked by another user. This optional argument can contain the following flag bit:

Symbol	Meaning
F.WAT	User will wait for access to the block

The INPUTL call defaults to not waiting if no flags are specified.

Special Devices

For devices that do not implement special processing of logical calls, the INPUTL call performs a READ call instead.

INPUTX - Perform a Logical Read Without Regard to Locking

The INPUTX call is identical to the INPUTL call, except that it returns a record even if the record is currently locked and even if it is being updated. INPUTX doesn't lock the record, and doesn't use the file locking flags. The calling sequence is:

```
INPUTX   adr{,flags}   ; get block from input device
```

adr references a DDB specifying the file being accessed and ***flags*** are ignored. See the description of INPUTL above for more information on using INPUTX.

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, OPENS, or OPENA) or random access (OPENR) before OUTPUT calls are performed. The calling sequence is:

```
OUTPUT   adr{,flags}   ; output block to output device
```

The OUTPUT call transfers the number of bytes in DDB+D.SIZ, but it normally does it as a standard block (depending on the driver in use). We discourage attempts to use the OUTPUT call for transferring non-standard record sizes.

Sequential File Processing

The main use of the OUTPUT call is in logical sequential file processing. The OUTPUT call starts its output based on the buffer index value in DDB+D.IDX. 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+D.BUF. For unit record devices this value is zero, since all data within the buffer is user data. For sequential disk files, however, the first part of each block within the file is a link to the next block; therefore, the value that should be in DDB+D.IDX is the size (in bytes) of the link, so output starts with the first non-link byte in the user buffer.

Since most output to sequential files and devices will be based on outputting bytes, words, or longwords, the usual method is to use the FILOTx calls (which use OUTPUT), rather than using OUTPUT directly. (See the Section "FILOTB, FILOTL, FILOTW - Output to a Device," below.)

Random File Processing

A special situation arises for files opened for random access by the OPENR call. Instead of writing the next sequential block, OUTPUT writes the specific relative block whose number is in DDB+D.REC from the user buffer.

You first set this number up and then execute the OUTPUT call. The block number is actually relative to the base of the file and has no direct relationship to the physical block on the device as would be written by a WRITE call.

The OUTPUT call requires the block being written previously was locked for exclusive use by an INPUTL call. If you are creating a new record that has not been read by an INPUTL call, use the OUTPTL call to write the block.

The optional flags argument can contain the following flag bit:

Symbol	Meaning
F.LOK	Output without unlocking

Special Devices

For devices that do not implement special processing of logical calls, the OUTPUT call performs a WRITE call instead.

OUTPTL - Perform a Logical Write With Locking

Except when used with files open for random access (OPENR), the OUTPTL call behaves exactly the same as the OUTPUT call. The calling sequence is:

```
OUTPTL   adr{,flags}   ; output block to output device
```

Sequential File Processing

The OUTPTL call works exactly like OUTPUT when used with sequential files.

Random File Processing

Like the OUTPUT call, OUTPTL writes the specific relative block whose number is in DDB+D.REC from the user buffer. The difference is that while the OUTPUT call requires the block in question was previously locked, OUTPTL does not require such a lock. OUTPTL is intended for use when creating new blocks and writing them for the first time.

The ***flags*** argument allows you to specify whether or not you wish your program to wait for access to the block, which may be locked by another user, or whether you want to write without unlocking the block. This optional argument can contain the following flag bits:

Symbol	Meaning
F.WAT	User will wait for access to the block
F.LOK	Output without unlocking

The OUTPTL call defaults to not waiting if no flags are specified.

Special Devices

For devices that do not implement special processing of logical calls, the OUTPUT call performs a WRITE call instead.

GET - Perform a Logical Record Read

The GET call allows you to read a single record from a file. Using the GET call on a file open for input by an OPENI call allows you to read sequential records, each terminated by a line-feed character. Using the GET call on a file open for record IO by an OPENIO call allows you to read random records within the file. The calling sequence is:

```
              GET     adr,flags,buff{,size}  ; get record from input device
```

Sequential File Processing

The GET call allows you to read a single record (line) from a sequential input file. The file you specify must be open for input (via OPENI) at the time you make the GET call.

The ***flags*** argument is not used for sequential file processing and should be omitted.

The ***buff*** argument must specify the address of a buffer into which the record will be read. This must ***not*** be the same as the buffer specified in D.BUF. A separate buffer must be supplied to hold the record.

The optional ***size*** argument allows you to specify a field into which the size of the record actually read will be placed. If you do not specify a size argument, the size will be returned in D6.

Prior to making the GET call, you must put the maximum allowable record size (i.e., the size of the buffer you specified) into D.RSZ. If the size of the record being read exceeds this maximum record size, the record will be truncated, with only as much data as will fit in the buffer being returned to the user. The next GET call will return the remainder of the record.

Sequential file records are considered to be any arbitrary sequence of ASCII characters up to and including a line feed character. These characters, including the line feed and a terminating null byte, are stored in the specified record buffer.

Random File Processing

For random files, which have been opened via an OPENIO call, the GET call behaves much like an INPUT call, except records are read instead of blocks. As with INPUT, you place the record number into D.REC prior to performing the GET call.

The GET call reads the requested file record but does not leave it locked, making the GET call the one to use when reading data that will not be updated. If you wish to update and rewrite the record you are reading, use the GETL call, which locks the record for exclusive use, allowing subsequent updates.

The ***flags*** argument allows you to specify whether or not you wish your program to wait for access to the record, which may be locked by another user, and what type of record blocking you are using, or whether you want to write without unlocking the record. The flags are described later in this section.

The ***buff*** argument must specify the address of a buffer into which the record will be read. This must ***not*** be the same as the buffer specified in D.BUF. A separate buffer must be supplied to hold the record.

The ***size*** argument is not used with random files record IO.

The flags argument can contain the following flag bits:

Symbol	Meaning
F.WAT	User will wait for access to the block
F.BIG	File uses records larger than 512 bytes and spans block boundaries
F.LOK	Write without unlocking

The GET call defaults to not waiting and AlphaBASIC compatible records if no flags are specified.

Special Devices

For devices that do not implement special processing of logical calls, the GET call performs a READ call instead.

GETL - Perform a Logical Record Read with Locking

The GETL call is identical to the GET call, except that when used with files open for record access (OPENIO), it leaves the requested record locked for exclusive use, allowing for subsequent update. The calling sequence is:

```
GETL     adr,flags,buff   ; get record from input device
```

Sequential File Processing

For sequential files, the GETL call is identical to the GET call.

Random File Processing

Like the GET call, GETL reads the specific relative record whose number is in D.REC into the user specified buffer.

The GETL call reads the requested file record and leaves it locked for exclusive use, allowing you to rewrite the record (via PUT) or unlock it (via UNLOKR). If you do not wish to update and rewrite the record you are reading, use the GET call, which does not leave the record locked.

The ***flags*** argument allows you to specify whether or not you wish your program to wait for access to the record, which may be locked by another user, and what type of record blocking you are using. The flags are described later in this section.

The ***buff*** argument must specify the address of a buffer into which the record will be read. This must ***not*** be the same as the buffer specified in D.BUF. A separate buffer must be supplied to hold the record.

The ***flags*** argument can contain the following flag bits:

Symbol	Meaning
F.WAT	User will wait for access to the block
F.BIG	File uses records larger than 512 bytes and spans block boundaries

The GETL call defaults to not waiting and AlphaBASIC compatible records if no flags are specified.

Special Devices

For devices that do not implement special processing of logical calls, the GETL call performs a READ call instead.

GETX - Perform a Logical Record Read Without Regard to Locking

The GETX call is identical to the GETL call, except that it returns a record even if the record is currently locked and even if it is being updated. GETX doesn't lock the record, and doesn't use the file locking flags. The calling sequence is:

```
        GETX     adr,flags,buff   ; get record from input device
```

adr references the DDB under which the file was opened, and ***flags*** are ignored. See the description of GETL for more information on using GETX.

PUT - Perform a Logical Record Write

The PUT call allows you to write a single record to a file. Using the PUT call on a file open for output via an OPENO call allows you to write sequential records, each terminated by a line-feed character. Using the PUT call on a file open for record IO via an OPENIO call allows you to write random records within the file. The calling sequence is:

```
        PUT      adr,flags,buff   ; put record out to device
```

Sequential File Processing

The PUT call allows you to write a single record (line) to a sequential output file. The file you specify must be open for output (via OPENO) at the time you make the PUT call.

The ***flags*** argument is not used for sequential file processing and should be omitted.

The ***buff*** argument must specify the address of a buffer from which the record will be written. This must ***not*** be the same as the buffer specified in D.BUF. A separate buffer must be supplied to hold the record.

Any size record may be written via the PUT call; however, to be able to read the record back from the file via a GET call, the record size must be restricted to 65535 bytes or less, including the line feed.

Sequential file records are considered to be any arbitrary sequence of ASCII characters up to and including a line feed character. These characters, including the line feed, are written to the output file.

Random File Processing

For random files, which have been opened via an OPENIO call, the PUT call behaves much like an OUTPUT call, except that records are written instead of blocks. As with OUTPUT, you place the record number into D.REC prior to performing the PUT call.

The PUT call writes a file record that has already been locked via a previous GETL call. Attempting to do a PUT to a record which is not locked will return an error. If you wish to write a record without having previously locked it, such as when creating a new file, use the PUTL call.

The ***flags*** argument allows you to specify whether or not you wish your program to wait for access to the record, which may be locked by another user, and what type of record blocking you are using. The flags are described later in this section.

The ***buff*** argument must specify the address of a buffer into which the record will be read. This must ***not*** be the same as the buffer specified in D.BUF. A separate buffer must be supplied to hold the record.

The flags argument can contain the following flag bits:

Symbol	Meaning
F.WAT	User will wait for access to the block
F.BIG	File uses records larger than 512 bytes and spans block boundaries

The PUT call defaults to not waiting and AlphaBASIC compatible records if no flags are specified.

Special Devices

For devices that do not implement special processing of logical calls, the PUT call performs a WRITE call instead.

PUTL - Perform a Logical Record Write with Locking

Except when used with files open for record access (OPENIO), the OUTL call behaves exactly the same as the PUT call. The calling sequence is:

```
        PUTL    adr,flags,buff  ; output record to output device
```

Sequential File Processing

The PUTL call behaves exactly like an PUT call when used with sequential files.

Random File Processing

Like the PUT call, PUTL writes the specific relative record whose number is in D.REC from the specified buffer. The difference is that while PUT requires the block in question was previously locked, PUTL does not require such a lock. PUTL is intended for use when creating new blocks and writing them for the first time.

The ***flags*** argument allows you to specify whether or not you wish your program to wait for access to the record, which may be locked by another user, and what type of record blocking you are using, or whether you want to write the record without unlocking it. The flags are described later in this section.

The ***buff*** argument must specify the address of a buffer into which the record will be read. This must ***not*** be the same as the buffer specified in D.BUF. A separate buffer must be supplied to hold the record.

The flags argument can contain the following flag bits:

Symbol	Meaning
F.WAT	User will wait for access to the block
F.BIG	File uses records larger than 512 bytes and spans block boundaries
F.LOK	Write without unlocking

The PUTL call defaults to not waiting and AlphaBASIC compatible records if no flags are specified.

Special Devices

For devices that do not implement special processing of logical calls, the PUTL call performs a WRITE call instead.

FILINB, FILINL, FILINW - Input from a Device

The FILINB, FILINL, and FILINW read a byte, a longword, and a word (respectively) from a file. These calls are the normal ones to use when handling sequential input files. All three calls take the address of a DDB as their argument, and return the result in D1.

The end-of-file condition may be tested for by seeing if DDB+D.SIZ is zero. The calling sequence is:

```
FILINx   adr           ; input from file
TST      adr+D.SIZ     ; end of file?
BEQ      eof           ;   branch if yes -
```

Your program may perform FILINW and FILINL on any byte boundary (that is, the buffer index does not need to be even). They read data from the file most significant byte first.

The unused high-order portion of D1 (on FILINB and FILINW) is cleared to zero.

FILOTB, FILOTL, FILOTW - Output to a Device

The FILOTB, FILOTL, and FILOTW write a byte, a longword, and a word (respectively) to a file. These calls are the normal ones to use when handling sequential output files. All three calls take the address of a DDB as their argument, and expect to find the item to be written in D1. The calling sequence is:

```
MOV      item,D1       ; get item to be output
FILOTx   adr           ; output to file
```

Your programs may perform FILOTW and FILOTL on any byte boundary (that is, the buffer index does not need to be even). They write data to the file most significant byte first.

In addition to the FILOTx calls, several other monitor calls perform file output, such as OUT, DCVT, OCVT, ERRMSG, and several others.

LOCKF - Lock a File

Use this command to relock a file which has already been OPENed. There are no defaults. This call is used by ISAM to lock index files. The calling sequence is:

```
LOCKF    adr,flags     ; lock the file
```

adr specifies a DDB specifying the file to be locked, and the ***flags*** argument can contain the following flag bits:

Symbol	Meaning
F.WAT	User will wait for access to the file
F.BIG	File uses records larger than 512 bytes and spans block boundaries
F.RON	Access in read only mode

(See the *AMOS File Locking Manual* for information on the different file locking modes.)

UNLOKF - Unlock a File

This command has the same format as CLOSE and is used to unlock a file without CLOSEing it. UNLOCKF is used by ISAM to unlock index files. The calling sequence is:

```
UNLOKF   adr,flags     ; unlock the file
```

adr specifies a DDB specifying the file to be unlocked and the ***flags*** argument can contain the following flag bits:

Symbol	Meaning
F.BIG	File uses records larger than 512 bytes and spans block boundaries

UNLOKR - Unlock a Record

This call unlocks a block record that was read with an INPUTL or GETL call but not yet written with an OUTPUT or PUT call. The calling sequence is:

```
UNLOKR   adr,flags     ; unlock the record
```

adr specifies a DDB specifying the file in which the record is to be unlocked and the ***flags*** argument can contain the following flag bits:

Symbol	Meaning
F.BIG	File uses records larger than 512 bytes and spans block boundaries

The D.REC field must contain the record or block number which is to be unlocked. For files open via OPENR, specify a block number; for files open via OPENIO, specify a record number.

DSKDEL - Delete a File

The DSKDEL call deletes a specific file from a file structured device. It takes the address of a DDB as its argument. The DDB must contain the filename, extension and PPN (if used) before executing the call. An error results if the file is not found. The calling sequence is:

```
DSKDEL   adr{,flags}   ; delete the file
```

adr references a DDB specifying the file to be deleted, and ***flags*** is an optional argument which can contain the following bits:

Symbol	Meaning
F.WAT	User will wait for access to the file

The DSKDEL call requires exclusive access to the file. It will default to not waiting for access if no flags are specified. DSKDEL closes the file before deleting it except for files opened with OPENI.

The DSKDEL call is ignored for devices which are not file structured.

DSKREN - Rename a File

The DSKREN call renames a specific file on a file structured device. It takes the address of a DDB as its argument. The DDB must contain the filename, extension and PPN (if used) before executing the call. The new filename and extension must be packed RAD50 into the three words immediately following the DDB in memory. The DSKREN call merely locates the directory item for the file and replaces the three words which store the filename and extension. The calling sequence is:

```
DSKREN   adr{,flags}   ; rename the file
```

adr references a DDB specifying the file to be renamed, and ***flags*** is an optional argument which can contain the following bits:

Symbol	Meaning
F.WAT	User will wait for access to the file

The DSKREN call requires exclusive access to the file. It will default to not waiting if no flags are specified. The DSKREN call is ignored for devices which are not file structured.

CHPROT - Change the Protection of a File

The CHPROT call changes the protection level of a specific file on a file structured device. It takes the address of a DDB as its argument. The DDB must contain the filename, extension and PPN (if used) before executing the call. The new protection must be placed in D.ARG prior to making the CHPROT call. The CHPROT call locates the directory item for the file and replaces the protection with the new one you specify.

The definition of how file protection is encoded may be found in the discussion of the D.PRT field found later in this chapter. The calling sequence is:

```
                CHPROT   adr{,flags}   ; change the file protection
```

adr references a DDB specifying the file to be changed, and ***flags*** is an optional argument which can contain the following bits:

Symbol	Meaning
F.WAT	User will wait for access to the file

The CHPROT call requires exclusive access to the file. It will default to not waiting if no flags are specified.

The CHPROT call is ignored for traditional file structures. It is also ignored for devices which are not file structured.

DSKCTG - Allocate a Contiguous File

The DSKCTG call is used to allocate a contiguous file on a random access device. It takes the address of a DDB as its primary argument. DSKCTG requires a second argument which represents the number of blocks to be allocated in the file. This second argument is passed in DDB+D.ARG. The calling sequence is:

```
                MOV      number of blocks,adr+D.ARG   ; set size into D.ARG
                DSKCTG   adr           ; try to allocate file
```

DSKCTG searches to find the first available area on the device which can fully contain the requested number of blocks. It marks these blocks as in use in the allocation bitmap, and adds a file descriptor item to the user directory. The word which gives the number of bytes in the last block is set negative to flag this file as contiguous, distinguishing it from the normal sequential files. A `device full` error results if no area on the device is large enough to contain the file.

If the devices are not file structured and random access, DSKCTG is ignored.

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. It takes the address of a DDB as its argument. Once a device has been assigned by this call, any attempt by another job to assign it results in an error. The device stays assigned to this job until de-assigned by the DEASGN call.

The calling format is:

```
                ASSIGN   adr           ; assign the device
```

The OPEN call automatically does an ASSIGN call, and the CLOSE call automatically does a DEASGN.

The ASSIGN call performs no action if the specified device is sharable, such as a disk.

DEASGN - De-assign a Device

The DEASGN call is used to de-assign a device which has been assigned to the user's job by the ASSIGN call. Once de-assigned, the device becomes available for assignment by other jobs. It takes the address of a DDB as its argument. The calling sequence is:

```
        DEASGN   adr           ; de-assign the device
```

The DEASGN call performs no action if the specified device is sharable or if it is not currently assigned to the user's job. All devices are de-assigned when the program exits to the monitor. A CLOSE call also performs a DEASGN.

DSKMNT - Mount a Disk Device

The DSKMNT call mounts a new disk device. Disk devices must be mounted before they can be accessed, and whenever a removable disk structure (floppy disk, disk cartridge, etc.) is changed. This mounting process is necessary for the system to make sure it is using the correct bitmap and other information pertaining to a particular device. In addition, many devices must be initialized before they can be used (such as when microcode must be loaded into the device controlled). Mounting the device signals it should be initialized.

DSKMNT accepts as arguments the address of a DDB which references the device you wish to mount and an optional flags argument. The DDB must have been previously INITed. The calling sequence is:

```
        DSKMNT   adr{,arg}     ; mount the disk
```

If MT$PHY is specified as the optional argument, a *physical* mount will be performed. If necessary, a physical mount call will "spin up" the disk drive.

On those devices that have alternate track allocation tables, a physical mount call also updates the alternate track table in memory. During this update process, DSKMNT checks the hash total of the BADBLK.SYS file. If this hash total is incorrect an error code is returned in D.ERR.

If MT$PHY is not specified, neither spin-up nor alternate track table updating is performed. A typical DSKMNT call might look like this:

```
        DSKMNT   @A5,#MT$PHY
```

Do NOT perform a DSKMNT call on a device other users are accessing. Doing so damages the disk's bitmap and can destroy the disk's files.

DSKUMT - Unmount a Disk Device

The DSKUMT call unmounts a disk device. Disk devices are unmounted to prevent further access to them, and before changing a removable disk and mounting a new one.

DSKUMT accepts as an argument the address of a DDB which references the device you wish to unmount. The DDB must have been previously INITed. A physical unmount call will "spin down" the disk drive if the specified device is the first logical unit of a physical Winchester disk drive. The calling sequence is:

```
            DSKUMT   adr          ; unmount the disk
```

Do NOT perform a DSKUMT call on a device other users are accessing. Doing so damages the disk's bitmap and can destroy the disk's files.

DISK SERVICE MONITOR CALLS

In the previous sections 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 consider the structure of the data. This section deals with the monitor calls used to manipulate that structure. You can find a description of the data structures used to maintain files on a device in Appendix A.

The disk presents special problems which require the use of special monitor calls to control the accessing of the directory and bitmap blocks. These blocks 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 blocks at the same time. The same holds true for the bitmap blocks. The following monitor calls are used to control the access to these non-sharable blocks:

Call	Meaning
DSKALC	Allocates the next available block on disk
DSKDEA	De-allocates a specific block on disk
DSKDRL	Sets re-entrant directory lock for a specific user
DSKDRU	Clears re-entrant directory lock for a specific user

The monitor routines normally access these blocks as a direct result of normal IO processing by file service calls. The process is somewhat tricky, and you should use the disk calls only 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.

Calling Sequence

All calls use the address of the relevant DDB as the standard argument. In addition to the first argument which is always the address of the DDB, some calls use an optional second argument for processing, which is passed in DDB+D.ARG. The second argument is detailed in the description of the individual call.

The Bitmap

Each file structured device uses a bitmap to perform its block allocation. This bitmap is stored on the device in a standard format, described in Appendix A. The structure in memory can take either of two forms: *paged*, where only the portion of the bitmap being accessed is stored in memory, or *non-paged* where the entire bitmap is stored in memory. Paged bitmaps are more efficient for large devices where the overhead of reading and writing the entire bitmap would be too great. Non-paged bitmaps are ideal for small devices (such as floppy disks) where the overhead of setting up for paging is greater than the reading and writing saved via paging.

Although the format of the bitmap in memory differs between paged and non-paged bitmaps, the format of the bitmap on the disk itself is always the same. The bitmap contains one bit for each logical block on the disk structure. This bit is off (0) if the block is free, and on (1) if the block is in use by anyone, including the system structure blocks themselves. Each word in the bitmap can define up to 16 blocks.

The first word in the bitmap defines blocks 0 through 17 (octal) with bit 0 defining block 0 and proceeding upward throughout the word. The second word defines blocks 20 through 37, and so on. To define the 500 decimal blocks in a standard IBM-compatible AMOS floppy disk, you need 32 words (32 times 16 = 512) with the last word not being totally used.

Following the allocation words in the bitmap are two words of hash total used for integrity validation. The bitmap for this device, therefore, takes up 34 words, including the two hash total words.

DSKALC - Allocate a Block

The DSKALC call allocates one block for use by the current user as a directory block or as a file block. You must specify as the argument the address of a DDB set up to reference the device on which you wish to allocate a block. DSKALC returns the block number of the allocated block in DDB+D.ARG. An error results if there are no free blocks left on the specified disk. The calling sequence for DSKALC is:

```
        DSKALC   adr          ; allocate the disk block
```

DSKALC performs a DSKBMR call first to insure the current job has access to the bitmap. The DSKALC call then locates the first free block and marks it in use. DSKALC flags the bitmap block as modified, causing it to be rewritten either at the next DSKBMW call or, if it must be swapped out to make room for another bitmap, sharing the same area in memory.

DSKDEA - De-allocate a Block

The DSKDEA call de-allocates a specific block on a disk and makes it immediately available for use by another user (or the same user). You must specify as the argument the address of a DDB set up to reference the device on which you wish to de-allocate a block. You must place the block number of the block you wish to de-allocate in DDB+D.ARG. The calling sequence is:

```
        DSKDEA   adr          ; de-allocate the disk block
```

The DSKDEA call does not check to see if this block is allocated to either the current user or any other user. First, DSKDEA performs a DSKBMR call to insure the current job has access to the bitmap, then it sets the specified block's bit to zero to indicate the block is free. Then it flags the bitmap block as modified to force a re-write.

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), DSKDEL, and for DSKREN calls. If the directory is already locked by another job, DSKDRL stalls until it is released. You must specify, as the argument, the address of a DDB referencing the device you wish to lock. The format is:

```
        DSKDRL   adr           ; lock the directory
```

The calling program or routine must unlock the directory via the DSKDRU call after the modifications have been made.

When performing directory accesses, you can alternately lock the directory through the use of the DIRACC call, discussed in Chapter 18.

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. You must specify, as the argument, the address of a DDB which references the device you wish to unlock. The calling sequence is:

```
        DSKDRU   adr           ; unlock the directory
```

Nothing happens if the directory is not locked by the current job.

MAGNETIC TAPE DRIVE MONITOR CALLS

There is a group of monitor calls designed to allow your assembly language program to access the various magnetic tape subsystems, including 1/2" 9-track tape drives, 1/4" cartridge streaming tape drives, and video cassette tape drives. Information on the software used with the 1/2" 9-track tape drives, VCRs, and streaming tapes, see your *System Commands Reference Manual*.

In addition to the magnetic tape monitor calls detailed below, you can use the READ and WRITE calls to input and output data to and from the tape units, in the same way you would perform physical disk I/0.

REWIND

This call issues a rewind command to the tape drive specified in the DDB. REWIND accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        REWIND   adr           ; rewind the tape
```

The DDB selects the device to which you want to issue a REWIND command. If an error results from this call, you may see the standard system file operation error messages.

If your VCR does not have computer control capability, The Video Cassette Recorder driver ignores this call.

WRTFM

This call issues a write-file-mark command to the specified tape unit. WRTFM accepts an argument that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        WRTFM    adr          ; write a file mark
```

The DDB selects the device to which you want to write a file mark. If an error results from this call, you may see the standard system file operation error messages.

FMARK

This call issues a find-file-mark command to the specified tape unit. FMARK will search for the file mark in a forward direction. FMARK accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        FMARK    adr          ; find a file mark, forward
```

The DDB selects the device to which you want to issue a find-file-mark command. If an error results from this call, you may see the standard system file operation error messages.

FMARKR

This call issues a find-file-mark command to the specified tape unit. FMARKR will search for the file mark in the reverse direction. FMARKR accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        FMARKR   adr          ; find a file mark, reverse
```

The DDB selects the device to which you want to issue a find-file-mark in reverse command. If an error results from this call, you may see the standard system file operation error messages.

BACKSP

This call issues a backspace command to the specified tape unit. BACKSP will back up over a single record on the tape. BACKSP accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        BACKSP   adr          ; move back one record
```

The DDB selects the device to which you want to issue a backspace command. If an error results from this call, you may see the standard system file operation error messages.

TAPSKP

This call will skip over multiple tape records on the specified tape unit. Not all tape drive/controller combinations support this call; those that don't ignore it. TAPSKP allows you to specify the number of records to skip, up to a maximum of 65535. TAPSKP accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        TAPSKP   adr           ; skip over some records
```

The DDB selects the device you want to issue a skip record command to. The number of records you wish to skip (up to 65535) is placed in the D.ARG field of the specified DDB. If an error results from this call, you may see the standard system file operation error messages.

TAPERS

This call performs an erase function. Not all tape drive/controller combinations support this operation; those that don't ignore this call. The call accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. It takes an additional argument in the D.SIZ field to specify the type of erase to be performed. The calling sequence is:

```
        TAPERS   adr           ; perform erase function
```

The DDB selects the device to which you want to issue an erase function to. The D.SIZ field specifies the type of erase function to be performed. If D.SIZ contains a zero, a fixed erase function will be performed. If D.SIZ contains a -1, a security erase of the entire tape will be performed (note this is a lengthy operation). Any other value in D.SIZ will cause that number of bytes to be erased on the tape, up to a maximum of 65535 bytes.

If an error results from this call, you may see the standard system file operation error messages.

UNLOAD

This call issues an unload command to the tape drive specified in the DDB. Not all tape drive/controller combinations recognize the UNLOAD call; those that don't ignore the call. UNLOAD accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        UNLOAD   adr           ; unload the tape
```

The DDB selects the device to which you want to issue an UNLOAD command. If an error results from this call, you may see the standard system file operation error messages.

RETNSN

This call issues a retention command to the tape drive specified in the DDB. Only the 1/4" Streaming Tape Drive driver recognizes RETNSN; all other magnetic tape drivers ignore it. RETNSN accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        RETNSN   adr           ; retention the streaming tape
```

The DDB selects the device to which you want to issue a retention command. If an error results from this call, you may see the standard system file operation error messages.

TAPST

This call issues a read-tape-status command to the specified tape unit. TAPST accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The DDB selects the device to which you want to issue a read-tape-status command. TAPST accepts a second argument which must be a register to which the tape status will be returned. The calling sequence is:

```
            TAPST    adr, reg      ; get tape status into register
```

The low-order 16 bits of the register will contain the tape status on completion of this call. The status symbols are defined as:

Symbol	Octal Value	Hex Value	Meaning
TP$7TK	1	1	Tape unit is a 7-track unit (1/2" 9-track magtape only).
TP$NRZ	2	2	Unit is in NRZI recording mode (1/2" 9-track magtape only).
TP$EOT	4	4	Unit has detected the end of the tape (1/2" 9-track magtape only).
TP$BOT	10	8	Tape is at the beginning of the tape.
	20	10	Tape unit is file protected (e.g., write ring is not installed).
TP$REW	40	20	Tape is currently rewinding.
	100	40	Tape unit is on-line.
	200	80	Tape unit is ready for a command.
ST$RDY	400	100	Tape unit is ready for a command (1/4" streamer and 1/2" 9-track magtape).
VC$DRR	400	100	VCR has data ready to be read (VCR interface only).
ST$EXC	1000	200	Tape unit has detected an exception (1/4" streamer tape only).
VC$DRW	1000	200	VCR is ready for a write command (VCR interface only).
ST$QIC	2000	400	Streamer supports QIC11/QIC24 commands

In addition to the symbols above, the upper five bits of the status word define the type of magnetic tape device. These five bits may be masked out using the value TP$DEV:

Symbol	Octal Value	Hex Value	Meaning
TY$MTU	0	0	Device is a 9-track magnetic tape drive.
TY$STR	20000	2000	Device is a 1/4" streaming cartridge tape drive.
TY$VCR	40000	4000	Device is a video cassette backup device.

Note many of these values are returned directly from the hardware interface. While all drivers make an attempt at maintaining device independence, some interfaces do not allow this because of the way in which they assert some of these signals. In particular, many 1/2" tape drives use the TP$7TK flag for purposes other than flagging 7-track operation, which is a very rare item today. Because the true meaning of this flag may be valuable, the tape drivers allow this bit to come through to the user issuing a TAPST call, even though the meaning is not device independent. Please consult the documentation for your particular tape device for a precise definition of the meaning of each of these bits.

TAPTYP

This call allows you to both select and inquire on the type of tape drive connected to the system. By selecting the drive type to match your hardware configuration, you allow the tape drive interface to optimize tape operations. TAPTYP accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        TAPTYP  adr         ; select tape type
```

The DDB selects the device on which you want to select or inquire on the tape driver type. To select a tape type, place the drive type code in D.ARG. To inquire as to which type is currently selected, place a -1 in D.ARG; the tape drive type will be in D.ARG after completion of the call. If after completion of the TAPTYP call D.ARG contains a -1, the TAPTYP call is not supported on the particular tape drive interface in use.

If an error results from this call, you may see the standard system file operation error messages.

TAPDEN

This call allows you to both select and inquire on the current recording density on a tape drive connected to the system. Most tape drives support multiple recording densities. TAPDEN accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        TAPDEN  adr         ; select recording density
```

The DDB selects the device on which you want to select or inquire on the tape density. To select a density, place the desired density (as bits per inch) in D.ARG. To select the default density (often selectable from the front panel of the tape drive itself) place a zero in D.ARG. After completion of the TAPTYP call, the actual density selected (which will be the closest available density to the value you supplied) will be in D.ARG.

To inquire as to the currently selected density, place a -1 in D.ARG; the current density will be in D.ARG after completion of the call. A zero returned in D.ARG indicates the default density. A -1 returned in D.ARG indicates this call is not available on the specified tape drive.

If an error results from this call, you may see the standard system file operation error messages.

TAPSPD

This call allows you to both select and inquire on the current transport speed on a tape drive connected to the system. Some tape drives support multiple transport speeds. TAPSPD accepts an address that references a DDB on which you have already performed an FSPEC, an INIT, and an OPEN monitor call. The calling sequence is:

```
        TAPSPD  adr         ; select transport speed
```

The DDB selects the device on which you want to select or inquire on the tape speed. To select a speed, place the desired speed (as inches per second) in D.ARG. To select the default speed (sometimes selectable from the front panel of the tape drive itself) place a zero in D.ARG. After completion of the TAPSPD call, the actual speed selected (which will be the closest available speed to the value you supplied) will be in D.ARG.

To inquire as to the currently selected speed, place a -1 in D.ARG; the current speed will be in D.ARG after completion of the call. A zero returned in D.ARG indicates the default speed. A -1 returned in D.ARG indicates this call is not available on the specified tape drive.

If an error results from this call, you may see the standard system file operation error messages.

Chapter 7
Terminal Service System

The Terminal Services (TRMSER) portion of the AMOS monitor has several calls which deliver data to and from both the user's terminal and other terminals connected to the system. Appendix B contains more information on the TRMSER routine itself and the various types of driver programs it uses to communicate with different types of terminals.

A terminal is defined as an ASCII character oriented device capable of both input and output. This is the formal definition and does not preclude the use of output only devices, such as printers, or input only devices, such as digitizing devices on terminal-designated I/O ports. Also, the system includes software terminals known as *pseudo terminals* which can be used to control jobs 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 executing the call. 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's terminal can be run without modification in a job controlled by a pseudo terminal. Keyboard input calls and terminal output calls always go to the controlling terminal, regardless of which job they are running in. Therefore, you 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.

The symbols used in this chapter are defined in TRM.UNV which is in DSK0:[7,7].

TERMINOLOGY

During the infancy of the computer industry, the most prevalent forms of terminals were teletype devices. Because of this, most terminal output calls reference the device name "TTY," a carry-over from those early days. And since the input device of the teletype was the keyboard, the input calls reference the device name of "KBD." These are strictly mnemonics and do not necessarily reflect the physical attributes of the terminals, which now are more commonly high-speed video display terminals.

THE TERMINAL CONTROL BLOCK

Each terminal has a *terminal control block* (TCB) associated with it in monitor memory. This table contains the parameters and work areas associated with the control of the terminal device. Most of the items in this TCB are for internal use only, and you need not be concerned with them. In fact, because the TCB is used to control the real time processing of terminal IO, modifying the contents of a TCB can have disastrous results.

The contents of the TCB, and the correct methods for use in accessing it, are described in Appendix B.

THE TERMINAL SERVICE CALLS

AMOS includes numerous monitor calls to perform input and output between the system and any of its connected terminals.

KBD - Fetch a Line of Data

The KBD call accepts one full line of input from the user terminal into a monitor line buffer, then sets index A2 to the base of that buffer for your reference. During the entry of the line, your 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.).

Enter a carriage-return, line-feed, or Control-C to terminate the line. The monitor automatically appends a line-feed to the carriage-return, and a null byte is set after the line-feed character. If you press CTRL/C, and the optional "label" has been specified, the program jumps to the specified label.

If the echo suppress flag is set in the terminal status word, it suppresses the normal echoing of the input characters, such as when the password is being entered for the LOG command.

For this form of KBD, the calling sequence is:

```
        KBD     {label}       ; get string; branch to label on Control-C
```

or:

```
        KBD                   ; get string
        {CTRLC label}         ; branch to label on Control-C
```

If the image mode input flag is set, the KBD command has a different effect. It performs no editing, and instead of accepting one line, it only accepts one character; that character is returned in register D1. (Register A2 is ***not*** set to the base of the monitor line buffer.) Image-mode input echoing is still under control of the echo suppress flag as in normal line mode.

In image mode, the calling sequence is the same as shown above.

TTY - Output One Character

The TTY call outputs one character from register D1 to the controlling terminal and then returns. Tabs are echoed as spaces up to the next modulo-8 carriage position, unless the data mode flag (T$DAT) is set in the terminal status word. If the job is running under the control of a command file, the character is only output to the terminal if the output suppress command is in the normal state (:R revives it, :S silences it). The calling sequence is:

```
        MOVB    char,D1       ; get character to type
        TTY                   ; type it
```

TIN - Get an Input Character

TIN gets the next input character from the terminal input buffer, and returns it in D1. Unlike the KBD call, TIN will not fetch input from a command file. This call is normally used only within the operating system itself and not by user programs. The calling sequence is:

```
        TIN                 ; D1 gets character
```

TOUT - Output One Character

TOUT outputs one character from D1 to the controlling terminal of the job. This call differs from the general TTY call in that the TOUT call does not check the command file status. The TOUT call, like the TIN call, is normally only used within the operating system itself. The calling sequence for TOUT is:

```
        MOVB    char,D1     ; get character to output
        TOUT                ; type it
```

TAB - Output One Tab

This convenience call outputs a single tab character to the user terminal. The calling sequence is simply:

```
        TAB                 ; display a tab
```

CRLF - Output a Carriage-Return/Line-Feed

This convenience call outputs a carriage-return and line-feed pair to the user terminal. The calling sequence is:

```
        CRLF                ; display carriage-return/line-feed
```

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. For example, you could use the following code to output two lines of data to the terminal:

```
        TTYI
        ASCII   /line 1 data/
        BYTE    15
        ASCII   /line 2 data/
        BYTE    15,0
        EVEN
```

The TTYI call also automatically appends a line-feed to all carriage-returns included in the string.

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 in line with the call itself in the program flow. TTYL takes one standard argument: the address of the message to be output. It 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 appends a line-feed to each carriage-return contained in the string. The calling sequence is:

```
        MOV     ptr to string,adr     ; get pointer to string to output
```

```
                TTYL     adr                    ; type the string
```

TCRT - Call Special Terminal Driver Routines

The TCRT call is the linkage into the special processing routine portion of a terminal driver. D1 usually contains a 2-byte code which is interpreted by the terminal driver routine as a special function, such as cursor positioning or special editing action. The only actions the TCRT call actually performs within TRMSER are locating the terminal driver for the attached terminal and calling the driver control routine within it. You must refer to the actual driver listing to determine the action performed in relation to the code passed to the driver in D1.

Standard Functions

The TCRT call is most commonly used for controlling such special CRT functions as cursor addressing and screen clearing. To maintain compatibility between terminal drivers, Alpha Micro has defined the following functions within the terminal drivers it supports.

Cursor Addressing

To perform cursor addressing, load D1 with a 2-byte argument defining the screen row and column to which the cursor is to be moved. The high-order byte should contain the row, and the low-order byte should contain the column. The uppermost-leftmost (home) position is row 1, column 1. The calling sequence for TCRT when used for cursor addressing is:

```
                MOVB     row,D1         ; get the row
                LSLW     D1,#8.         ; shift up to high order
                MOVB     column,D1      ; move in column
                TCRT                    ; position the cursor
```

Other Functions

To perform other special CRT functions, bits 8-15 of D1 should be loaded with a negative number, usually a -1. The negative number in this field informs the terminal driver that a special terminal function is to be performed.

Most of the special terminal functions you will use require a -1 in bit 8-15. A wide variety of different functions are available, ranging from clearing the terminal screen to drawing special characters and shapes. The list given below is not complete, but is provided here for easy reference. See the *AMOS Terminal Service System User's Guide* for a complete and up-to-date listing of available functions, as well as a complete description of each call's behavior.

The calling sequence for TCRT when used for the functions listed below is:

```
                MOVW     #<-1_8.>+func,D1 ; move function & flag to D1
                TCRT                      ; execute the function
```

Then load the low-order byte with one of these special decimal function codes. Calls marked with an asterisk (*) are either obsolete calls, or calls for special purposes. We recommend they not be used, as they may change in the future.

Low-byte Value (Decimal)	Meaning
0	Clear screen and set normal intensity
1	Cursor home (move to 1,1)
2	Cursor return (move to column 1 without line-feed)
3	Cursor up one row
4	Cursor down one row
5	Cursor left one column
6	Cursor right one column
7	Lock keyboard
8	Unlock keyboard
9	Erase to end of line
10	Erase to end of screen
11	Enter background display mode (reduced intensity)
12	Enter foreground display mode (normal intensity)
13	Enable protected fields
14	Disable protected fields
15	Delete line
16	Insert line
17	Delete character
18	Insert character
19	Read cursor address
20	Read character at current cursor address
21	Start blinking field
22	End blinking field
23	Start line drawing mode (enable alternate character set)
24	End line drawing mode (disable alternate character set)
25 *	Set horizontal position
26 *	Set vertical position
27	Set terminal Attributes
28	Cursor on
29	Cursor off
30	Start underscore
31	End underscore
32	Start reverse video
33	End reverse video
34	Start reverse blink
35	End reverse blink
36	Turn off screen display
37	Turn on screen display
38	Top left corner
39	Top right corner
40	Bottom left corner
41	Bottom right corner
42	Top intersect
43	Right intersect
44	Left intersect
45	Bottom intersect
46	Horizontal line
47	Vertical line
48	Intersection
49	Solid block
50	Slant block
51	Cross-hatch block
52	Double line horizontal
53	Double line vertical

Low-byte Value (Decimal)	Meaning
54	Send message to function key line
55	Send message to shifted function key line
56	Set normal display format
57	Set horizontal split (follow with row code)
58 *	Set vertical split (39-character columns)
59 *	Set vertical split (40-character columns)
60 *	Set vertical split column to next character
61	Activate split segment 0
62	Activate split segment 1
63	Send message to host message field
64	Up arrow
65	Down arrow
66	Raised dot
67	End of line marker
68	Horizontal tab symbol
69	Paragraph
70	Dagger
71	Section
72	Cent Sign
73	One-Quarter
74	One-Half
75	Degree
76	Trademark
77	Copyright
78	Registered
79	Print screen
80	Set to wide (132 column) mode
81	Set to normal (80 column) mode
82	Enter transparent print mode
83	Exit transparent print mode
84	Begin writing to alternate page
85	End writing to alternate page
86	Toggle page
87	Copy to alternate page
88	Insert column
89	Delete column
90	Block fill with attribute
91	Block fill with character
92	Draw a box
93	Scroll box up one line
94	Scroll box down one line
95	Select jump scroll
96	Select fast smooth scroll
97	Select medium-fast smooth scroll
98	Select medium-slow smooth scroll
99	Select slow smooth scroll
100	Start underscored, blinking field
101	End underscored, blinking field
102	Start underscored, reverse field
103	End underscored, reverse field
104	Start underscored, reverse, blinking field
105	End underscored, reverse, blinking field
106	Start underscored text without space
107	End underscored text without space
108	Start reverse text without space
109	End reverse text without space
110	Start reverse blinking text without space

Low-byte Value (Decimal)	Meaning
111	End reverse blinking text without space
112	Start underscored blinking text without space
113	End underscored blinking text without space
114	Start underscored reverse text without space
115	End underscored reverse text without space
116	Start underscored reverse blinking text without space
117	End underscored reverse blinking text without space
118	Start blink without space
119	End blink without space
120	Set cursor to blinking block
121	Set cursor to steady block
122	Set cursor to blinking underline
123	Set cursor to steady underline
124	Reserved
125	Reserved
126	Reserved
127	Reserved
128	Select top status line without address
129	End status line (all kinds)
130	Select unshifted status line without address
131	Select shifted status line without address
132 *	Select black text
133 *	Select white text
134 *	Select blue text
135 *	Select magenta text
136 *	Select red text
137 *	Select yellow text
138 *	Select green text
139 *	Select cyan text
140 *	Select black reverse text
141 *	Select white reverse text
142 *	Select blue reverse text
143 *	Select magenta reverse text
144 *	Select red reverse text
145 *	Select yellow reverse text
146 *	Select green reverse text
147 *	Select cyan reverse text
148	Save a rectangular area
149	Restore a rectangular area
150	Enable full graphics mode
151	Disable full graphics mode
152	Draw box with rounded corner
153	Draw "window" style box
154	Draw double-line box
155	Enable proportionally spaced text
156	Disable proportionally spaced text
157	Select color palette
158	Enable graphics cursor
159	Disable graphics cursor
160	Select graphics cursor style
161	Inquire graphics cursor position
162	Define graphics cursor region
163	Form feed symbol
164	Line feed symbol
165	New line symbol

Low-byte Value (Decimal)	Meaning
166	Vertical tab symbol
167	Plus-or-minus symbol
168	Greater-than-or-equal symbol
169	Less-than-or-equal symbol
170	Not-equal symbol
171	British pound sign
172	Pi character
173-255	Reserved

Not all terminal drivers have all of the above functions, simply because not all terminals have all of these features. Be sure to check the availability of a call (by using TRMCHR) before using the TCRT call.

All TCRT codes are reserved for future expansion by Alpha Micro. If you have a specific need for a new TCRT function, you may contact Alpha Micro Software Development to reserve a TCRT code number. Using this reservation process avoids problems of conflicting and incompatible TCRT functions.

Color

TCRT codes with the high order byte set to -2 or -3 are used for color control on terminals that support color. To select a foreground color, a TCRT call should be executed with the high order byte set to -2, and the low order byte selecting the desired color from the list below. To select a background color, use a high-order byte of -3 and a low-order byte of the desired color.

The colors that may be used in the low-order byte are listed below. Note that while only eight colors are currently defined, all 8 bits of the low order byte are reserved for color definition, allowing up to 256 colors. The first 8 colors are the primary colors, and the remaining colors will have reduced saturation and lightness as appropriate, according to the HLS (Hue, Lightness, Saturation) color model. Details on the assignment of colors can be found in the *AMOS Terminal Service System User's Guide.*

Low-byte Value (Decimal)	Color
0	Black
1	White
2	Blue
3	Magenta
4	Red
5	Yellow
6	Green
7	Cyan

This will allow a program to be written for a 256 color terminal that could be made to work on an 8 color terminal by reducing the requested color back to the closest primary color in the terminal driver, probably by setting the Lightness to 50%, the Saturation to 100%, and rounding the Hue to the closest color.

As an example of setting the color, the following code will select yellow text on a blue background:

```
MOVW     #<-2_8.>+5,D1 ; select FOREGROUND + YELLOW
TCRT                   ; send the command
MOVW     #<-3_8.>+2,D1 ; select BACKGROUND + BLUE
```

```
               TCRT                ; send the command
```

Terminals which do not support color will ignore all TCRT calls with the high-order byte set to -2 or -3, allowing programs which use color commands to also work on monochrome terminals without modification.

On terminals which support it, the color palette (the selection of available colors) can be changed through the use of the TCRT -1,157 call.

Additional Features

Many terminals contain additional features, such as proportionally spaced text and variable numbers of rows and columns, which can also be accessed through the TCRT monitor call. These additional features use TCRT calls with a negative high-order byte.

Details on these additional features can be found in the *AMOS Terminal Service System User's Guide.*

RTCRT - Perform a Remote TCRT Call

The TCRT monitor call (described above) provides a terminal independent method of accessing terminal control functions within the terminal attached to the job issuing the TCRT call. Certain applications, however, need the ability to perform terminal control functions on terminals not directly attached to the job. The RTCRT monitor call described in this section provides a method for issuing such remote terminal control functions.

As with the TCRT call, D1 is loaded with the TCRT function number prior to issuing the call. The function numbers used by RTCRT are identical to those used by TCRT.

The calling sequence for RTCRT is:

```
               MOVW     function,D1   ; select function to perform
               RTCRT    port          ; send the command
```

The ***port*** argument specifies the address of the TCB associated with the terminal on which the terminal control function is to be performed. You may locate this address by using the TCBIDX monitor call.

TCKI - Check for Input

This call checks to see if there are any characters waiting in the user's terminal input buffer. If there is at least one character, the Z-bit is set; if not, the Z-bit is cleared. The calling sequence is:

```
               TCKI                   ; is there any input waiting?
               BEQ      input         ;   yes - there is at least one
```

TRMCHR - Get Terminal Characteristics

This call returns a brief description of the terminal attached to your job in a standardized form. Different terminals have widely varying characteristics, requiring each program to adapt itself to the particular terminal in use. The ability to use a wide variety of terminals is an important feature of AMOS based software.

While most of the various terminal functions are entirely independent of a particular terminal's features, there are times when it is desirable for a program to find out details about the terminal, such as the width or height of the screen.

The format for calling TRMCHR is:

```
          TRMCHR   adr,flags
```

adr points to an argument formatted as described below, and ***flags*** is a combination of bit flags which affect the operation of the TRMCHR call (also described below).

The argument block pointed to by adr is formatted:

0 2	Terminal flags	TC.FLG
4	Number of rows on screen	TC.ROW
6	Number of columns on screen	TC.COL
10	Number of available colors	TC.CLR
12	Current foreground color	TC.FGC
14	Current background color	TC.BCG
16	Number of rows in window	TC.WNR
20	Number of columns in window	TC.WNC
22	Length of top status line	TC.TSL
24	Length of unshifted bottom status line	TC.USL
26	Length of shifted bottom status line	TC.SSL
30	Maximum number of characters in saved area	TC.SVA
32	Feature bitmap	TC.BMP

TC.FLG is a 32-bit field containing flags describing the various characteristics and features of the terminal. These are the same flags used in the terminal driver and are defined in SYSSYM. It is the responsibility of each program to check for the availability of a terminal feature before using it. Failure to do so will result in unpredictable results.

The list below is only a partial listing of the flags in use. See the *AMOS Terminal Service System User's Manual* for a complete and up-to-date list of the flags, as well as the precise meaning of each flag.

Symbol	Meaning
TD$ALP	Has alternate page
TD$AMS	Is an Alpha Micro terminal
TD$BLF	Has block fill
TD$BLN	Has blink
TD$BOX	Has "box" commands
TD$CID	Has character insert/delete
TD$CLR	Has color capability
TD$DIM	Has dim
TD$EOL	Has erase to end of line
TD$EOS	Has erase to end of screen
TD$EXT	Has support for 8-bit extended character sets
TD$KID	Has column insert/delete
TD$LID	Has line insert/delete
TD$MOD	Is a "mode" terminal, rather than a "field" terminal (displays attributes only when written)
TD$MLT	Has multi-key (function key) translation
TD$NSP	Has "no space" attribute commands (TCRT -1,106-119)
TD$PHR	Has AM-70 color commands (TCRT -1,132-147)
TD$PRT	Has printer support
TD$RVA	Has reverse video
TD$SMT	Has smooth scroll
TD$SPL	Has horizontal split screen
TD$STS	Has status line
TD$UND	Has underscore
TD$132	Has 80/132 column support
TD$GRA	Has full graphics capability
TD$VRC	Has variable number of rows and/or columns
TD$PRO	Has proportionally spaced text capability
TD$GRY	Has monochrome gray-scale capability

The flags allowed on the TRMCHR call are:

Symbol	Meaning
TC$BMP	The TRMCHR call will return a bitmap describing the TCRT functions available on the terminal, in addition to other information returned. The bitmap contains a single bit for each TCRT function. If a bit is set to 1, that TCRT function is available. If set to 0, it is not. Bits are numbered starting with the low-order bit of the first word being 0, corresponding to the TCRT -1,0 function. The bitmap is returned in an area corresponding to TC.BMP. It contains 256 bits.
TC$CMP	The TRMCHR call will return the current contents of the color map. If this flag is not specified, no color map will be returned. Reserved for future use.

To make it easy to allocate the space required to hold the returned arguments, three symbols have been defined:

- The first, TC.SIZ, defines the number of bytes returned by TRMCHR without the optional color map or optional "bitmap."
- The second, TC.SZC, defines the number of bytes returned by TRMCHR with the optional color map.
- The third, TC.SZB, defines the number of bytes returned by TRMCHR with the optional "bitmap."

Here is an example that uses the TRMCHR call to retrieve all of the available terminal information (both the standard information and the bitmap):

```
GETIMP   TC.SZB,A5     ; get impure area for returned info
TRMCHR   @A5,TC$BMP    ; get the information
LEA      A1,TC.BMP(A5) ; A1 now indexes the bitmap
```

Message Calls

Three calls have been defined in SYS.UNV as macros using the TTYI call. These calls are for your convenience and make your the programs easier to understand. They all take a single argument which is an ASCII message string to be output to the user terminal. Due to the way macro arguments are processed, if the message has leading or trailing spaces, or if it has embedded commas, you must enclose it with angle brackets or part of it will be lost. The three calls are:

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

Note the message may not contain any slashes (/), since these are used as delimiters for the ASCII statement in the macros.

TRMRST - Read Terminal Status

This monitor call allows you to read the terminal status word associated with a terminal. This 16-bit field has certain flags in it you may modify to alter the operation of your terminal calls. The terminal status word has the following flag positions defined:

Symbol	Octal Value	Hex Value	Meaning
T$IMI	1	1	Set to force image mode input (see KBD call)
T$ECS	2	2	Set to suppress echoing of input characters
T$LCL	4	4	Set if terminal has local echoing (half-duplex)
T$DAT	10	8	Set to engage data mode to allow complete data transparency on input and output (^C, nulls, and 8-bit characters are all passed through)
T$ILC	20	10	Set to allow lower-case input (disables conversion)
T$XLT	40	20	Set to allow multi-key sequences for function key translation
T$NFK	100	40	Disables all function key processing—overrides T$XLT
T$OIP	200	80	Set if output is in progress (internal flag only)
T$LED	400	100	Set if monitor line editor is in use by this terminal
T$ASN	1000	200	Set if this terminal port is assigned
T$DIS	2000	400	Set if this terminal port is disabled
T$VLD	4000	800	Set if T.POO field contains a valid value
T$LDT	10000	1000	Set if in special line editor mode
T$EXT	20000	2000	Enable 8-bit extended character sets
T$OSP	40000	4000	Output has been suspended (XOFF)
T$JLVL	100000	8000	Output data processing to be done at job level

Because the terminal status word is a resource that must be shared among multiple processors, it is important you use the TRMRST and TRMWST calls to access it. Direct access to this field (or any other field with the TCB) can cause interprocessor deadlock, leading to system failure. The calling sequence for TRMRST is:

```
        TRMRST   dst{,port}
```

The ***dst*** argument receives the 16-bit terminal status word. The optional ***port*** argument allows you to specify the address of a TCB whose status word you wish to read. If you do not specify a second argument, TRMRST will return the status of the terminal attached to the executing job. You can determine a terminal's TCB address via the TCBIDX monitor call.

TRMWST - Write Terminal Status

This monitor call allows you to write the terminal status word associated with a terminal. This 16-bit field has certain flags in it you may modify to affect the operation of various terminal functions. A description of the contents of the terminal status word is given in the previous section. The calling sequence is:

```
        TRMWST   src{,port}
```

The ***src*** argument provides the 16-bit value you wish written to the terminal status word. The optional ***port*** argument allows you to specify the address of a TCB whose status word you wish to write. You can determine a terminal's TCB address via the TCBIDX monitor call. If you do not specify a second argument, TRMWST will write the status to the terminal attached to the executing job.

The TRMWST call will most frequently be used immediately after a TRMRST call, as the most common operation is to read the current status, update one or more bits, and then to write it back to the terminal status word. To simplify this process, and to eliminate a common error in software which must manipulate the terminal status word, the TRMWST call does ***not*** allow the setting or resetting of the T$OIP flag. The T$OIP flag can only be manipulated by using the TINIT monitor call.

The monitor resets the TIMI, TECS, T$DAT, and T$ILC bits in the terminal status word each time your program exits back to AMOS command mode, thereby restoring normal terminal operation regardless of what your program may set the terminal status flags to.

TTYIN - Fetch Another Job's Input

The TTYIN call allows one job to get waiting input data from the terminal input buffer of another job. The character is returned in D1; A5 must index the TCB for the terminal you wish to fetch data from. The calling sequence is:

```
        MOV      tcb ptr,A5    ; get pointer to other jobs terminal
                 ;             ;  definition table entry
        TTYIN                  ; get character into D1
```

TTYOUT - Place a Character in Another Job's Output

The TTYOUT call allows one job to put data into another job's terminal output buffer. It is used by the TRM device driver to perform the output function.

The character you wish to output must be in D1; A5 must index the TCB for the terminal you wish to output to. The calling sequence is:

```
MOV      tcb ptr,A5    ; get pointer to other job's terminal
MOVB     #character,D1 ; get character to output
TTYOUT                 ; send the character
```

The terminal you output to does not have to be attached to a job for this call to work.

TRMICP - Process Input Character Within Interface Driver

The TRMICP call is executed from within a terminal interface driver to process one character which the hardware interface has just received from the terminal. D1 must contain the input character to be processed, and A5 must index the TCB for the specific terminal being serviced. TRMSER then takes the character and passes it to the terminal driver input routine for pre-processing if desired.

When the terminal driver passes it back, TRMSER edits the character for control codes and other special characters, then adds it to the terminal input buffer. All the pertinent flags are set automatically to indicate 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 is awakened to process the available data. Register D1 is modified.

The calling sequence is:

```
MOV      tcb ptr,A5    ; get pointer to our TCB into A5
MOVB     char,D1       ; get character into D1
TRMICP                 ; hand character to TRMSER
```

Note TRMICP is also used to implement the FORCE program. If A5 is set to index the TCB of another job's terminal, TRMICP will cause the character in D1 to be placed in that terminal's input buffer.

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 to be sent to the terminal. This is usually in response to an interrupt from the interface board, indicating the prior character has been fully output and the board is ready to transmit the next character. Register A5 must index the TCB for the specific terminal being serviced, and D1 gets the next available character upon return from TRMSER processing of the call. If there is no more output available in the output buffer, D1 is set to -1 as a flag, and the associated job is awakened to fill the output buffer again.

The calling sequence is:

```
MOV      tcb ptr,A5    ; get pointer to our TCB into A5
TRMOCP                 ; get character into D1
TSTW     D1            ; any characters available?
BMI      none          ;   no-
```

You can also use TRMOCP to get the output characters from a job whose terminal interface driver and terminal driver have both been defined as PSEUDO. This can be done by pointing A5 to the other job's terminal's TCB. Note this use of TRMOCP will only function correctly if the other job's interface driver (.IDV) is PSEUDO, and the terminal driver (.TDV) is ***not*** NULL.

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 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 this call is used, D1 must index the buffer to be queued or contain the character to be output, D3 must contain the number of characters in the buffer, and A5 must index the TCB entry for the specific terminal. The TRMBFQ call performs the output initiation function if the output system for the terminal is currently idle.

The calling sequence is:

```
         MOV     tcb ptr,A5    ; index the TCB
        MOV     count,D3      ; get character count
        MOV     string ptr,D1 ; index output string
        TRMBFQ                ; queue up the output
```

For example, it is often necessary for a terminal driver to output a backspace-space-backspace character sequence in response to a rubout. The TRMBFQ call should be used to output this sequence. In the example shown below we assume A5 already indexes the terminal's TCB (as it does within a terminal driver).

```
        MOV     #3,D3         ; get number of characters to output
        LEA     A6,ERUB       ; index the characters to output
        MOV     A6,D1         ; get into proper register
        TRMBFQ                ; queue up the characters
        RTN                   ; return
ERUB:   BYTE    10,40,10,0    ; backspace-space-backspace
```

Another function commonly performed within terminal drivers is the outputting of nulls, used to delay output during lengthy terminal operations such as clear screen on a CRT, or carriage-return on a teleprinter.

The example below outputs 20 filler characters (octal 200). This example also assumes A5 has previously been set up.

```
        MOV     #20.,D3       ; output 20 nulls
        MOV     #200,D1       ; a null is an octal 200
        TRMBFQ                ; output the characters
        RTN                   ; return to caller
```

TBUF - Output Large Amounts of Data

TBUF is the normal call your programs can execute to queue up large amounts of data into the terminal output system of a terminal where the single character calls are considered in-efficient. It is a buffered call, working through the two output buffers for the terminal, instead of going directly into the output queue system. If you try to output more data via the TBUF call than there is currently room for in the output buffers, the job is suspended while the output buffers are unloaded to the terminal. Each time one of the output buffers empties, the job awakens and the TBUF call proceeds to fill that buffer. This continues until the original amount of data is exhausted, at which time TBUF returns to the calling program.

Note this call is efficient only for large amounts of data, not for short strings.

When the call is executed, A2 must index the buffer to be output and D3 must contain the number of characters to be output (similar to the TRMBFQ call).

The calling sequence is:

```
MOV     char count,D3   ; get number of chars to output
MOV     char index,A2   ; index the string to be output
TBUF                    ; output them
```

TCBIDX - Index a Terminal Control Block

When you need to gain access to another terminal's TCB, you can use the TCBIDX monitor call to locate that TCB. You specify the job whose TCB you wish to index by supplying TCBIDX with an ASCII string giving the terminal name. The TCBIDX call returns the base address of the TCB in the destination argument you provide. Further access to the TCB should be done by using this base address as an index.

If the specified terminal is located successfully, the Z-bit will be set upon completion of the call. If no terminal with the specified name can be located, the call returns with the Z-bit reset. Note this assumes the use of an address register as the destination argument. Use of memory as the destination will corrupt the return flags.

The calling format is:

```
TCBIDX  string,dst
BNE     term-not-found
```

The following code will locate the TCB for the terminal named *TERM1* and display its address on the terminal.

```
        TCBIDX  NAME, A1        ; index TCB with A1
        BNE     NOTFND          ;   error - TCB not found
        MOV     A1,D1           ; get into register to display
        OCVT    0,OT$TRM        ; display the address
        CRLF                    ; make it pretty
        EXIT                    ;   and return to AMOS

NOTFND: TYPECR  <?Unable to locate terminal>
        EXIT

NAME:   ASCIZ   /TERM1/         ; name of terminal to find
        EVEN
```

Chapter 8
Conversion Monitor Calls

AMOS provides a series of monitor calls devoted to converting data from one format to another. Included are calls to convert binary to ASCII, ASCII to binary, ASCII to packed RAD50, etc. In addition to the calls defined in this chapter, Chapter 11 describes additional calls to convert to and from the various floating point formats.

NUMERIC CONVERSION CALLS

The AMOS monitor contains two calls which perform conversions from a single binary longword value to an ASCII formatted decimal or octal string. Conversion options allow you to send the string to the terminal, to an output file, or to a buffer in memory. Options also let you control the format of the result.

Calling Format

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

```
DCVT    size,flags    ; Convert binary number in D1 to decimal
OCVT    size,flags    ; Convert binary number in D1 to octal
                      ;  (hex if J.HEX is set for this job)
```

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 is 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 D1 value is zero, at least one zero digit will be output as the result.

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:

Symbol	Octal Value	Hex Value	Meaning
OT$ZER	1	1	Disables leading zero blanking
OT$TRM	2	2	Outputs the result to the user terminal
OT$DDB	4	4	Outputs the results to the file whose DDB is indexed by A2
OT$MEM	10	8	Puts the result in memory at the buffer indexed by A2 and updates A2
OT$LSP	20	10	Adds one leading space to the result
OT$TSP	40	20	Adds one trailing space to the result

Note that the maximum value you can display using these calls is the maximum value of a 32-bit word. All numbers are considered unsigned so the largest decimal number is 4,294,967,295; the largest octal number is 37777777777; and the largest hex number is FFFFFFFF.

If the size byte is non-zero, the effect of the leading-zero blanking flag described above is reversed. That is, when the size byte is zero, the conversion calls default to leading zero blanking, with OT$ZER turning that blanking off. When the size byte is non-zero, the calls default to leading zeros, with OT$ZER specifying that leading zeros are to be blanked.

The following examples may clarify things a bit. All examples assume the value in D1 is 964 (decimal), and the symbol · in the result field indicates a blank.

Call	Displayed Result
DCVT 0,OT$TRM	964
DCVT 0,OT$TRM!OT$LSP	·964
DCVT 0,OT$TRM!OT$TSP	964·
DCVT 5,OT$TRM	00964
DCVT 5,OT$TRM!OT$ZER	··964
DCVT 5,OT$TRM!OT$ZER!OT$TSP	··964·
DCVT 5,OT$TRM!OT$LSP!OT$TSP	·00964·
DCVT 2,OT$TRM	64 (the 9 is lost)

RAD50 CONVERSION MONITOR CALLS

Radix-50 (RAD50) packing is a system by which three ASCII characters may be packed into a single 16-bit word using a special algorithm based on the value of octal 50. Radix-50 packing is used throughout the system where the packing of filenames and other data entities lends itself. 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 (Octal)
blank	0
A-Z	1-32
a-z	1-32
$	33
. (period)	34
%	35
0-9	36-47

RAD50 Packing Algorithm

The packing algorithm for a 3-character input to a 16-bit RAD50 result is:

1. The first character code is multiplied by 3100 octal (50 x 50).
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. See Appendix G, "RAD50 Conversion Table," for information on converting data manually between octal and RAD50.

Packing and Unpacking Calls

There are two monitor calls which perform the above packing and unpacking algorithms. Both calls use registers A1 and A2 as indexes to the components and require no calling arguments.

PACK - Pack Three ASCII Characters into RAD50

The triplet (three ASCII characters) indexed by A2 is packed into RAD50 form, and the result is left in the word indexed by A1. A1 is incremented by two to receive the next result word for multiple packing. A2 is left indexing the first character, which was not included in the packing of this triplet. The PACK call terminates packing and forces blank fill for any input which does not contain three valid RAD50 characters. For the PACK call, a blank is considered an illegal input character and terminates packing.

The calling sequence for PACK is as follows:

```
MOV      buff addr,A1  ; index the RAD50 buffer
MOV      ascii addr,A2 ; index the ASCII string
PACK                   ; convert ASCII to RAD50
```

UNPACK - Unpack Three RAD50 Characters into ASCII

The word in the address indexed by A1 is unpacked, and the triplet is left in the three bytes beginning with the byte currently indexed by A2. A1 is incremented by two for the next word, and A2 is incremented by three for the next triplet result. Blanks are legal in unpacking and are placed into the result if they are decoded from the input word.

```
MOV      rad50 addr,A1 ; index the RAD50 string
MOV      buff addr,A2  ; index the ASCII buffer
UNPACK                 ; convert RAD50 to ASCII
```

PRINTING CONVERSION CALLS

Three calls in the monitor 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.

PFILE - Type a Filespec from a DDB to the Terminal

The argument specifies the address of a file DDB, and the PFILE call extracts the parameters in the file specification words. It then prints them on the user terminal in the standard format of dev:filnam.ext[p,pn]. The account is not printed if it is the same as the user's current login. Calling sequence:

```
        PFILE   adr          ; print the file specification
```

OFILE - Output a File Specification

Provides a flexible method for converting full or partial file specifications from internal RAD50 packed format (from the DDB) to an ASCII character string. Allows you to direct the ASCII output to your terminal, to memory, or to an output file (by using a DDB). In addition, OFILE allows you to specify which portions of the file specification you wish to convert, allowing you to output only a device, a device and PPN, or a full file specification. The format is:

```
        OFILE   ddb, flags
```

ddb points to a standard DDB containing the file specification to be output, and ***flags*** contains binary flags specifying the destination of the ASCII text and the portions of the filespec to be converted:

Symbol	Octal Value	Hex Value	Meaning
OT$TRM	2	2	Output the specification to the user's terminal
OT$DDB	4	4	Output the specification to the DDB indexed by A2
OT$MEM	10	8	Output the specification to memory, using the pointer contained in A2
OT$LSP	20	10	Output a leading space before the filespec
OT$TSP	40	20	Output a trailing space after the filespec
OT$OFD	100	40	Output the device specification
OT$OFN	200	80	Output the file name and extension
OT$OFP	400	100	Output the PPN if not current login

Note that if OT$OFD, OT$OFN, and OT$OFP are all set to zero (that is, no portion of the file specification is selected) OFILE will output the entire specification.

However, if the user is logged into the PPN specified in the DDB, the PPN is not displayed unless it is the only part of the specification to be output. That is, the current PPN is output only if OT$OFP is specified and OT$OFD and OT$OFN are not specified.

Here is an example of how to use OFILE to output a file specification to the memory buffer BUFF(A5):

```
        LEA     A2,BUFF(A5)   ; Load effective address into A2
        OFILE   DDB(A4),OT$MEM!OT$OFD!OT$OFN!OT$OFP
```

PRNAM - Output a Filename

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

```
        PRNAM   adr           ; print the filename
```

PRPPN - Output a PPN

The argument addresses a 1-word project-programmer code, and the PRPPN call prints the converted result on the user terminal in the standard format of proj,prog. The p,pn is output in octal, regardless of the setting of J.HEX. The calling sequence is:

```
        PRPPN   adr           ; print the ppn
```

VCVT - Output a Version Number

The VCVT monitor call is provided as a convenient way of unpacking and displaying the version number contained in the program header area. (The program header area is defined via the PHDR macro, discussed in Appendix A. Information on the packed format of the version number may also be found there.) The format is:

```
        VCVT    adr,flags     ; convert version number at adr
                              ;  to a string
```

adr points to a 32-bit packed version number (see Appendix A for the packed format), and ***flags*** are defined:

Symbol	Octal Value	Hex Value	Meaning
OT$TRM	2	2	Outputs the result to the user terminal
OT$DDB	4	4	Outputs the results to the file whose DDB is indexed by A2
OT$MEM	10	8	Puts the result in memory at the buffer indexed by A2 and updates A2
OT$LSP	20	10	Adds one leading space to the result
OT$TSP	40	20	Adds one trailing space to the result

These flags are (intentionally) the same as those for DCVT, OCVT, FCVT, and ERRMSG.

For example, to display the version number at address ***VER*** on the user's terminal, you would specify:

```
        VCVT    VER,OT$TRM    ; display version number
```

CASE CONVERSION CALLS

Two monitor calls are provided for converting from lower to upper case and from upper to lower case.

UCS - Convert Lower to Upper Case

This monitor call converts the character in D1 to upper case. Only alphabetic (a-z) characters are affected. If the conversion resulted in the character changing, the Z-flag is set; otherwise, it is reset. Calling sequence:

```
MOVB    char,D1       ; get the character
UCS                   ; convert to upper case
```

The high-order three bytes are not modified.

LCS - Convert Upper to Lower Case

This monitor call converts the character in D1 to lower case. Only alphabetic (A-Z) characters are affected. If the conversion resulted in the character changing, the Z-flag is set; otherwise, it is reset. Calling sequence:

```
MOVB    char,D1       ; get the character
LCS                   ; convert to lower case
```

The high-order three bytes of register D1 are not modified.

Chapter 9 Input Line Processing Calls

When an operator command executes a program, register A2 is left pointing to the first non-blank character on the command line which follows the command name itself. The command program normally interprets the remainder of the line and uses it to determine the files to act on, the record number to dump, the devices to access, etc. For example, the M68 call requires the name of the program and any switch options to follow the M68 command name on the same line. The macro assembly program then processes the program name and the switch options by way of the A2 index which indexes the rest of the command line. This command line is actually the user's terminal input buffer.

Not only does the command input line use A2, the KBD monitor call also leaves A2 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 A2 to the base of that line for scanning. For this reason, a number of monitor calls exist which perform scanning and conversion functions based on an input line which is indexed by A2. Some of the calls merely test the character indexed by A2 for a specific condition and return with flags set, based on the result of the test. In these instances A2 is not modified. In calls which perform scan conversions, A2 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 D1.

ALF - TEST A CHARACTER FOR ALPHABETIC

The ALF call tests the character indexed by A2 to determine whether it is alphabetic (A-Z; a-z); the Z-flag is set if it is, and cleared if it is not. A2 is not changed. Calling sequence:

```
        ALF                     ; is character alphabetic?
        BEQ      label          ;   yes -
```

NUM - TEST A CHARACTER FOR NUMERIC

The NUM call tests the character indexed by A2 to determine whether it is numeric (0-9); the Z-flag is set if it is, and cleared if it is not. A2 is not changed. Calling sequence:

```
        NUM                     ; is character numeric?
        BEQ      label          ;   yes -
```

TRM - TEST A CHARACTER FOR TERMINATOR

The TRM call tests the character indexed by A2 to determine if it is a legal terminator defined as a blank, tab, comma, semicolon, carriage-return, line-feed, or null. The Z-flag is set if the character is a terminator, and cleared if it is not. A2 is not changed. Calling sequence:

```
        TRM                     ; is character a field terminator?
        BEQ      label          ;   yes -
```

LIN - TEST A CHARACTER FOR LINE TERMINATOR

The LIN call tests the character indexed by A2 to determine if it is a legal end-of-line character defined as a semicolon, carriage-return, line-feed, or null. The Z-flag is set if it is an end-of-line character, and cleared if it is not. A2 is not changed. Calling sequence:

```
LIN                    ; is character a line terminator?
BEQ      label         ;   yes -
```

BYP - BYPASS BLANKS

The BYP call advances index A2 past all characters which are blanks or tabs, and leaves it indexing the first non-blank, non-tab character it finds. Calling sequence:

```
BYP                    ; skip over blanks
```

GTDEC - INPUT A DECIMAL NUMBER

The GTDEC call uses index A2 to process a decimal number whose value may be from 0 to 4,294,967,295 in the input line (leading zeros are legal), and to deliver the resulting binary value back in D1. The N-flag is set if there is an error (i.e., result is greater than 4,294,967,295). GTDEC updates A2 to point to the character following the decimal input number. In case of an error, A2 is left indexing the digit that would have caused the overflow past 4,294,967,295 (useful for multiple precision processing techniques). Calling sequence:

```
GTDEC                  ; read in number
BMI      error         ;   error if too large
```

GTOCT - INPUT AN OCTAL NUMBER

The GTOCT call uses index A2 to process an octal number whose value may be from 0 to 37777777777 in the input line (leading zeros are legal), and to deliver the resulting binary value back in D1. The N-flag is set if there is an error (i.e., result is greater than 37777777777). GTOCT updates A2 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 processes input in hexadecimal instead of octal (the maximum number then being hex FFFFFFFF). Calling sequence:

```
GTOCT                  ; read in number
BMI      error         ;   error if too large
```

GTPPN - INPUT A PROJECT-PROGRAMMER NUMBER

The GTPPN call uses index A2 to process a project-programmer number in the standard format of proj,prog, and to deliver the resultant binary code back in D1. The format dictates that project numbers be octal numbers between 1 and 377, and that programmer numbers be octal numbers between 0 and 377.

The N-flag is set if the PPN was not in valid format. GTPPN updates A2 to point to the character following the PPN. Calling sequence:

```
GTPPN                ; read in ppn
BMI     error        ;   error if invalid format
```

FILNAM - INPUT A FILENAME

The FILNAM call uses index A2 to process a filename.extension input string, leaving the RAD50 packed 3-word result in the three words starting with the address specified as the first argument of the call. In format, this argument is a standard monitor call argument. The second argument is a 1- to 3-character extension to be used in case no explicit extension is entered in the input string. FILNAM updates A2 to index the terminating character. The Z-bit is set if there was no filename to process (i.e., the first character was not a legal RAD50 character). Calling sequence:

```
FILNAM  adr,default extension  ; read in filename to adr
BEQ     error        ;   error if no filename to process
```

Chapter 10
Date and Time Conversion Calls

Alpha Micro computer systems contain a battery backed-up clock/calendar, providing the system with an accurate reading of the current time and date. This clock/calendar is supported through a series of monitor calls, which isolate the program from the different hardware implementations used to provide the clock/calendar service. These calls allow you to both read and set the time and date. This chapter discusses the calls and the data formats used to represent time and date within AMOS.

In addition to the calls discussed below, the system library, SYSLIB.LIB, contains several utility routines to accept and display dates and times. See Appendix D for further information.

GDATES - GET DATE IN SEPARATED FORMAT

This call returns the current date in separated format (Month-Day-Year) in the specified destination argument. The calling sequence is:

```
           GDATES   dst          ; get date
```

The result is formatted as:

Day	Month	Day of Week	Year
31 24	23 16	15 8	7 0

In this illustration, day is 1-31, month is 1-12 for January to December, day of week is 0-6 for Monday to Sunday, and year is the year biased by –1900. Therefore 1980 will return 80 (decimal) and 2010 returns 110 (decimal).

GDATEI - GET DATE IN INTERNAL FORMAT

This call returns the current date in internal format in the specified data register. Internal format is a true Julian date, where the date is a 32-bit integer corresponding to the number of solar days since Greenwich noon of the last concurrence of:

- The 4-year leap year cycle
- The 7-year solar cycle
- The 19-year Metonic cycle (235 lunations = 19 years—2 hours)
- The 15-year indiction cycle (Roman taxation interval).

Using this format, a value of zero corresponds to January 1, 4713 BC at 12:00 GMT. This format makes it quite easy to determine the day of the week or the number of days between two dates. It is also useful for historians as it predates just about all known events to the exact date, making all historical dates positive Julian dates.

The call accepts a standard destination argument. The format is:

```
              GDATEI   dst          ; get system date
```

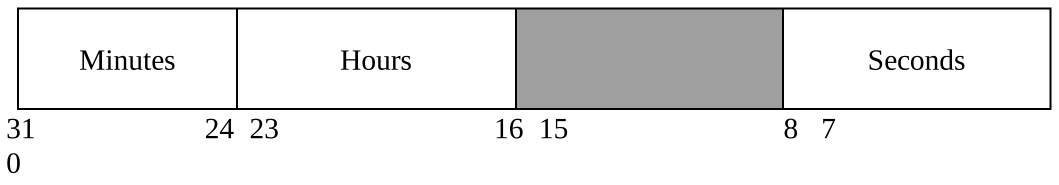
The date returned by this call may be packed into a 16-bit quantity by converting it to number of days since January 1, 1900 12:00 GMT. This may be done by simply subtracting 2415021 (decimal) from the quantity returned by GDATEI.

GTIMES - GET TIME IN SEPARATED FORMAT

This call returns the current time of day in separated format in the specified destination argument. The calling sequence is:

```
              GTIMES   dst          ; get system time
```

The result is formatted as:



The hours, minutes, and seconds are binary numbers. Hours are in 24-hour format.

GTIMEI - GET TIME IN INTERNAL FORMAT

This call returns the current time of day in internal format (seconds since midnight) in the specified destination. The calling sequence is:

```
              GTIMEI  dst
```

SDATES - SET SYSTEM DATE FROM SEPARATED FORMAT

The SDATES monitor call allows you to set the system date. The call accepts one argument—the date you wish to set—in separated format. (Separated format is discussed above, under the GDATES monitor call.) The calling sequence is:

```
              SDATES   src          ; set the system date
              BNE      error        ; branch if error
```

You may set the system date only if you are logged into [1,2]. If you attempt to set the date while logged into any other PPN, the SDATES call returns with the Z-flag off. If the call successfully sets the system date, it sets the Z-flag on.

See the section "Setting the System's Clock/Calendar," below, for more about SDATES.

STIMES - SET SYSTEM TIME FROM SEPARATED FORMAT

The STIMES monitor call allows you to set the system time. The call accepts one argument—the time you wish to set—in separated format. (Separated format is discussed above, under the GTIMES monitor call.) The calling sequence is:

```
STIMES   src          ; set the system time
BNE     error         ; branch if error
```

You may set the system time only if you are logged into [1,2]. If you attempt to set the time while logged into any other PPN, the STIMES call returns with the Z-flag off. If the call successfully sets the system time, it sets the Z-flag on.

Note that some systems, in particular the AM-100/L, AM-1000, AM-1200, AM-1500, and AM-2000, do not allow the seconds field to be set. This is a limitation of the clock/calendar integrated circuit used on these systems. These systems will use whatever the current value of the seconds field is when executing an STIMES monitor call.

YEAR 2000 ISSUES

AMOS has a number of features to assist with the changeover from the 20th to the 21st century. The problem which these features (if they are used correctly in an application) help to solve is the interpretation of two-digit years when the four-digit year changes from 19xx to 20xx. The AMOS system is geared to handle this changeover; the date in the system's clock/calendar chip will roll over correctly from December 31, 1999 to January 1, 2000, and AMOS library routines and monitor calls will reflect this change.

AMOS Date Formats

AMOS distinguishes between three classes of dates:

- Dates used to set the system's clock/calendar chip
- Extended directory format date information
- All other dates

AMOS manipulates dates in three formats:

- Internal date format is not affected by the century changeover, as the internal date is held as the number of days since January 1, 4713 BC on a proleptic calendar.
- Separated date format is defined with the year value being held as one byte, as described earlier in this chapter. The value in this byte is the number of years since 1900. Therefore, 1996 would be represented by a decimal year value of (1996 - 1900) = 96, and the year 2010 would be represented a decimal year value of (2010 - 1900) = 110. With the year held as a single byte, the

absolute maximum range of years in separated format is from 1900 to (1900 + 255) = 2155. Some date manipulation algorithms may handle only a subset of these separated date year values.

- Extended directory format dates use separated date format as a starting point for the packing and unpacking algorithms implemented in the $PAKDT and $UNPDT library calls, but use only 7 bits to hold the year value. Thus they can hold dates in the range of 1900 to 2027 inclusive.

AMOS Date Conversion Routines

For converting ASCII date representations to separated date format, AMOS supplies two routines: $IDTIM and $IDTIMX. Both routines deal with two-digit and four-digit year strings, but in different ways. $IDTIM treats a two-digit year as being in the 20th century (i.e., as a year 19xx). Otherwise it expects to find a four-digit year value in the range of 1900 through 2155 inclusive. Any other values will cause the routine to set an error flag. $IDTIMX implements the same rules as setting the system date: a two-digit year in the range 80 through 99 is treated as the year 1980 through 1999 inclusive, and values of 0 through 79 are treated as the year 2000 through 2079 inclusive. A four-digit year must be in the range of 1900 through 2155 inclusive. Like $IDTIM, values outside this range will set an error flag.

For converting separated date representations to ASCII strings, AMOS supplies two routines, $ODTIM and $ODTM2. Both routines deal with years in the same way. A value of 1900 is added to the year byte. If a four-digit year is requested, all four digits are displayed. If a two-digit year is requested, only the last two digits of the result are shown.

Setting the System's Clock/Calendar

The system's clock/calendar chip is set by using the SDATES call, which takes a date in separated format and sets the system's date appropriately. Due to the design of many calendar chips, especially those made many years before 2000, there is no place to store the century: the year is stored as two digits. This means that you cannot set the date to read it back unambiguously: if you read back the system date after setting it to 1/1/1960 or to 1/1/2060, you get 1/1/60 without any century information. As the system date is active, being automatically updated at midnight every day, this poses a problem as the year rolls over from 1999 to 2000. How does AMOS tell programs that the year is now 2000, not 1900?

AMOS 2.3 solves this problem transparently to all programs. AMOS restricts the range of years to which the system date can be set to the years 1980 through 2079 inclusive. You cannot set the system date to a date outside this range. To set the system date, you use the SDATES monitor call. Its longword parameter must have the year byte set to 80. (representing 1980) through 179. (representing 2079) inclusively. Any other value will produce undefined results. You must set the date explicitly once after upgrading to AMOS 2.3, or after downgrading to an earlier version of AMOS. After that initial setup of the calendar chip, AMOS date processing will function correctly.

Retrieving the system date through the GDATES call will return a year byte in the same range of values (80. through 179. inclusively). If the date was set correctly, no other values can be retrieved.

Note that this restriction in range for setting and retrieving the system date only affects system date setting and retrieving. Separated date format is still valid over the range 0 (representing 1900) through 255 (representing 2155) inclusively as far as storing values are concerned. Remember that date manipulation algorithms may have a more restricted range.

Chapter 11
Floating Point Monitor Calls

AMOS provides a complete set of monitor calls supporting three different floating point data types. Calls are provided for floating point arithmetic (addition, subtraction, multiplication, and division), for conversion to and from binary, and for conversion to and from ASCII text. Through the use of these calls you can avoid needing to know the intricacies of floating point arithmetic and treat floating point as just another data type available to the assembly language programmer.

AMOS supports the use of three different floating point data types. The first is a 48-bit format unique to Alpha Micro. This format provides slightly better than 12 digits of accuracy within floating point operations. AMOS also supports two floating point formats conforming to the IEEE standard format for floating point numbers. The first, a 32-bit single precision format, provides 8 digits of accuracy. The second, a 64-bit double precision format, provides 15 digits of accuracy.

The Alpha Micro 48-bit format is compatible with all Alpha Micro computer systems, and is in use within thousands of existing applications, making it the most common format used in Alpha Micro software. It is implemented as a series of software routines within AMOS.

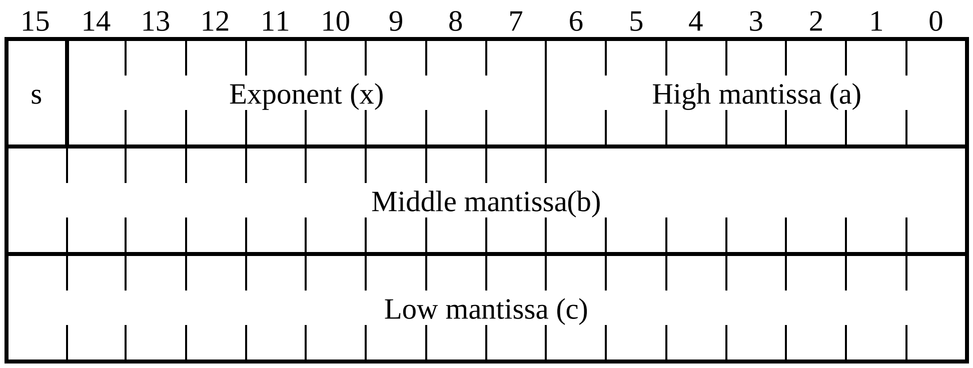
The IEEE format 32- and 64-bit formats are compatible with a wide variety of other computer systems. The availability of greater precision in the 64-bit format also makes it attractive. If floating point hardware is available on a given system, that hardware will be used for all computation. If such hardware is not available, AMOS will use software floating point routines to implement the IEEE format support.

ALPHA MICRO 48-BIT FLOATING POINT FORMAT

AMOS supports a 48-bit floating point format consisting of a 40-bit mantissa, 8-bit exponent, and a sign-bit. Because a normalized mantissa always starts with a binary one in the high-order bit, this bit is not stored in the floating point number but is always assumed to be present. This allows us to represent the 40-bit mantissa in only 39 bits.

The floating point format supported by AMOS is fully compatible with the floating point representation used by the AM-100 and AM-100/T processors.

Floating point numbers are represented in three 16-bit words formatted as follows:



The values of s, x, a, b, and c (considered as unsigned integer fields) determine a floating point number "f" according to the following formulas:

```
if (x > 0) then
    f(s,x,a,b,c) = (if s = 0 then 1 else -1)
                 · (1 + a 2^{-7} + b 2^{-23} + c 2^{-39})
                   2^{x-129}
```

if $(x = 0)$ and $(s, a, b, \text{or } c \neq 0)$ then f(s,x,a,b,c) is undefined

if $(x = 0)$ and $(s, a, b, \text{and } c = 0)$ then f(s,x,a,b,c) is zero

Given the definitions above, note that you can correctly compare a floating point number to zero by comparing only the first word to zero.

When negating a floating point number: if the number is zero, leave it unchanged; otherwise, complement the sign bit.

Floating Point Arithmetic

AMOS supports four calls to perform 48-bit format floating point arithmetic: FADD, FSUB, FMUL, and FDIV, which perform addition, subtraction, multiplication, and division, respectively.

These four calls, and the FCMP call, all have one level of indirection built into the source and destination operands. Thus, the instruction FADD A3,A5 does not add the numbers in A3 and A5, but rather adds the two numbers pointed to by A3 and A5 (just as if they had been in an ADD @A3,@A5 instruction).

Because of the way these calls are implemented, you may not use SP as an operand to any of the floating point monitor calls (e.g., FADD SP,A5 is illegal).

FADD - Floating Point Add

The FADD monitor call performs 48-bit format floating point addition. It adds the floating point number pointed to by the source operand to the floating point number pointed to by the destination operand and stores the result in the location pointed to by the destination operand. The overflow condition is cleared, and other condition codes are set by testing the high word of the result.

The calling sequence for FADD is:

```
        FADD    src,dst
```

FSUB - Floating Point Subtract

The FSUB monitor call performs 48-bit format floating point subtraction. It subtracts the floating point number pointed to by the source operand from the floating point number pointed to by the destination operand and stores the result in the location pointed to by the destination operand. The overflow condition is cleared, and other condition codes are set by testing the high word of the result.

The calling sequence for FSUB is:

```
        FSUB    src,dst
```

Note that unlike the FSUB instruction on the AM-100 and AM-100/T processors, the AMOS FSUB monitor call does ***not*** negate the source operand in memory.

FMUL - Floating Point Multiply

The FMUL monitor call performs 48-bit format floating point multiplication. It multiplies the floating point number pointed to by the source operand by the floating point number pointed to by the destination operand and stores the result in the location pointed to by the destination operand. The overflow condition is cleared, and other condition codes are set by testing the high word of the result.

The calling sequence for FMUL is:

```
        FMUL    src,dst
```

FDIV - Floating Point Divide

The FDIV monitor call performs 48-bit format floating point division. It divides the floating point number pointed to by the source operand into the floating point number pointed to by the destination operand and stores the result in the floating point number pointed to by the destination operand. The overflow condition is cleared, and other condition codes are set by testing the high word of the result.

The calling sequence for FDIV is:

```
        FDIV    src,dst
```

If the source operand is zero, a divide by zero exception occurs. See Section 11.1.5 for further information.

Floating Point Conversion

AMOS supports five conversion calls to convert 48-bit format floating point numbers to and from longwords and extended words, and to ASCII.

FFTOL - Floating Point to Longword Conversion

The FFTOL monitor call truncates an AMOS format floating point number to a two’s complement 32-bit integer. If the floating point number is greater than the largest 32-bit integer or less than the smallest 32-bit integer, the least significant 32 bits of the resulting integer will be returned. The calling format is as follows:

```
FFTOL    src,dst
```

Note that only memory effective addresses are valid as source operands.

FLTOF - Longword to Floating Point Conversion

The FLTOF monitor call converts a two's-complement 32-bit integer to 48-bit format floating point. The calling format is as follows:

```
FLTOF    src,dst
```

Note that only memory effective addresses are valid as destination operands.

FFTOX - Floating Point to Extended Conversion

The FFTOX monitor call truncates an AMOS format floating point number to a 40-bit two's-complement integer. If the floating point number is greater than the largest 40-bit integer or less than the smallest 40-bit integer, the most significant 40 bits of the resulting integer (all 40 bits of the mantissa) are returned. This call preserves more precision than the FFTOL call. The calling format is as follows:

```
FFTOX     src,dstl,dsth
```

src points to the floating point number to be converted, ***dstl*** specifies where the low-order longword is to be stored, and ***dsth*** specifies where the high-order byte of the result is to be stored.

Note that only memory effective addresses may be used to specify the source operand.

FXTOF - Extended to Floating Point Conversion

The FXTOF monitor call converts a 40-bit two's-complement integer to 48-bit format floating point. This call preserves more precision than the FLTOF call. The calling format is as follows:

```
FXTOF    srcl,srch,dst
```

srcl specifies the low-order longword of the source, ***srch*** specifies the high-order byte of the source, and ***dst*** specifies where the floating point result is to be stored.

Note that only memory effective addresses may be used to specify the destination operand.

FFTOA - Floating Point to ASCII Conversion

The FFTOA monitor call converts a 48-bit format floating point number to a packed ASCII representation of that number. FFTOA is intended for use by runtime routines that must perform formatting and editing before displaying the result. For a more convenient way to display floating point numbers, see the FCVT monitor call (Section 11.1.3.3).

The calling format of FFTOA is:

```
FFTOA   src,dst
```

src specifies where to get the floating point number and ***dst*** specifies the location of a 14-byte buffer. This 14-byte buffer is formatted as one 16-bit word containing the decimal exponent, and 12-bytes containing ASCII characters representing the decimal mantissa.

You may obtain a valid display of the floating point number by displaying the sign of the number, a decimal point, the characters in the decimal mantissa, an "E," and the signed integer contained in the decimal exponent.

Note that only memory effective addresses may be used as the source and destination operands.

FFTOAX - Floating Point to ASCII Extended Conversion

The FFTOAX monitor call converts a floating point operand to 13 decimal digits. It operates the same as the FFTOA Floating Point to ASCII conversion call except that a 15-byte buffer instead of a 14-byte buffer must be provided by the src pointer, and the dst pointer must point to an area at least 13 bytes long.

The calling format of FFTOAX is:

```
FFTOAX  src,dst
```

src specifies where to get the floating point number and ***dst*** specifies the location of a destination buffer.

FFTOAX has been provided because the 40-bit mantissa of an AMOS floating point number provides somewhat more than 12 decimal digits of precision, and being able to convert a 13th decimal digit may provide a better indication of the value of the number for very precise calculations. The accuracy of the thirteenth digit will depend on the value of the two most significant digits of the converted number. The higher the value of these two digits, the less the information contained in the 13th digit. When the first two digits are "10," the thirteenth digit is fully precise.

Floating Point Input/Output Calls

AMOS supports three floating point input/output calls: one to get a floating point number from a memory buffer, one to get a floating point number from a I/O device, and one to output floating point numbers.

GTFLT - Get a Floating Point Number

The GTFLT monitor call converts an ASCII representation of a 48-bit format floating point number to internal floating point format. GTFLT uses A2 as a pointer to the ASCII input string, and stores the result in the specified destination. It updates A2 to point to the character following the floating point number. A floating point number is terminated by the first character that is not a legal continuation of the number.

The calling format is:

```
GTFLT    dst
```

Note that only memory effective addresses are valid as the destination operand.

The format of the ASCII input consists of an optional sign ("+" or "-"), up to twelve significant decimal digits with an optional embedded decimal point ("."), and an optional exponent. If an exponent is specified, it must consist of an "E" followed by an optional sign ("+" or "-") and the exponent value itself (in the range ± 38). For example:

```
1       3.14159265357       1E38
1.0     -1.123E-12          +12.4
```

You may use the OT$NLD flag with GTFLT to disable language definition files, if the characters caused by the language interfere with the read. The format is:

```
GTFLT    @A5,OT$NLD
```

GTFLTF - Get a Floating Point Number from a File

The GTFLTF monitor call performs the same function as the GTFLT call, except that it reads its input string from a file whose DDB is indexed by A2, rather than from a memory buffer.

The calling format (with A2 indexing a DDB), is as follows:

```
GTFLTF  dst
```

Note that only a memory effective address may be used as the destination operand.

You may use the OT$NLD flag with GTFLTF to disable language definition files, if the characters caused by the language interfere with the read. The format is:

```
GTFLTF   @A5,OT$NLD
```

FCVT - Output a Floating Point Number

The FCVT monitor call returns an ASCII representation of a 48-bit format floating point number to memory, to a file, or to the user's terminal. It outputs the number in the standard format shown below. For more extensive formatting of floating point numbers, use the FFTOA call described in Section 11.1.2.5.

FCVT is called in the following format:

```
        FCVT    src,size,flags,precision,scale
```

The operand fields are defined as follows:

src — A memory effective address specifying where the floating point number to be displayed is to be fetched from.

Size — Specifies the minimum width of the field to be displayed. The number appears right justified within the field, padded with leading spaces.

Flags — Contains formatting flags defined as follows:

Symbol	Meaning
OT$FIX	Fixed format. "Precision" actually contains the number of digits to print after the decimal point.
OT$TRM	Output the result to the user's terminal.
OT$DDB	Output the result to the file whose DDB is indexed by A2.
OT$MEM	Output result to memory buffer indexed by A2 and update A2.
OT$LSP	Add one leading space to the result.
OT$TSP	Add one trailing space to the result.
OT$NSP	Don't replace the leading "-" with a space on positive numbers.
OT$NLD	Disable all use of language definition files.
OT$SCI	Force output to scientific notation in the same format as used when the value is too small or large to output in the default format.

Results are undefined when both OT$SCI and OT$FIX are used.

Precision — If bit 0 of flags is a zero, then precision specifies how many decimal digits to round to before displaying the number (the default is 11). If bit 0 of flags is a one, then precision specifies how many decimal digits to display after the decimal point (the default is zero).

Scale — Specifies a decimal scaling factor. The floating point number is divided by the scaling factor prior to being output. The default scaling factor is zero.

Three output formats are possible: scientific notation ("sx.xxxEsxx"), decimal notation ("sxxx.xxx" — variable length fraction), and fixed notation ("sxxx.xxx00" — rounded/zero padded fraction). Scientific notation is used when the number is too large or too small to be displayed in decimal notation, or too large to be displayed in fixed notation. When fixed notation is specified, a precision of 11 is used for scientific notation.

Miscellaneous Floating Point Calls

In addition to the calls described above, AMOS also supports two miscellaneous calls: FCMP and FPWR, used for comparing two 48-bit format floating point numbers and for scaling floating point numbers, respectively.

FCMP - Floating Point Compare

The FCMP call compares two 48-bit format floating point numbers and sets the condition codes appropriately. Note that the two floating numbers compared are those *pointed to by the source and destination operands, not the operands themselves.*

```
              FCMP     src,dst
```

FCMP sets the condition codes for a *signed* branch.

FPWR - Floating Point Multiply by a Power of Ten

The FPWR call multiplies a 48-bit format floating point number by a power of ten. This operation is frequently used when performing scaled arithmetic.

The calling format is:

```
              FPWR     src,arg
```

The source argument, which must be a memory effective address, is multiplied by the power of ten specified by the argument. The power of ten may be either positive or negative.

For example, to multiply the floating point number pointed to by A5 by 1000 (10^3), you would specify:

```
              FPWR     @A5,#3
```

To multiply the same number by 1/1000 (10^{-3}):

```
              FPWR     @A5,#-3
```

Floating Point Error Trapping

During certain 48-bit format floating point operations, error conditions can occur. These error conditions are division by zero, numeric overflow, and numeric underflow. AMOS provides a method of trapping these errors so that your program can take corrective action.

If you do not specify that these errors are to be trapped, AMOS executes its default error processing routines which display an appropriate error message on your terminal and abort the program back to AMOS command level.

If you wish instead to handle these errors yourself, you should place the address of your error processing routine in JOBFPE in your job's JCB. (Further information on JCBs and how to access them may be found in Chapter 2.) Then, when an error occurs, AMOS executes your error processing routine.

When your error processing routine is called, the condition codes are set as follows:

Condition	Meaning
V = 1 and N = 0	Floating underflow has occurred
V = 1 and N = 1	Floating overflow has occurred
V = 0 and N = 1	Divide by zero was attempted

A6 then points to the floating point number that caused the error. Your routine may modify this number if desired. For example, you might want to handle an underflow condition by returning a zero. Your error routine must preserve all registers and return with the condition codes set for the returned floating point result (TSTW @A6 will accomplish this).

If your trap routine simply executes a RTN instruction, default values are used as the result. For underflow, a value of 0.0 is supplied; for overflow and divide by zero, the largest positive number is used (approximately 1E38).

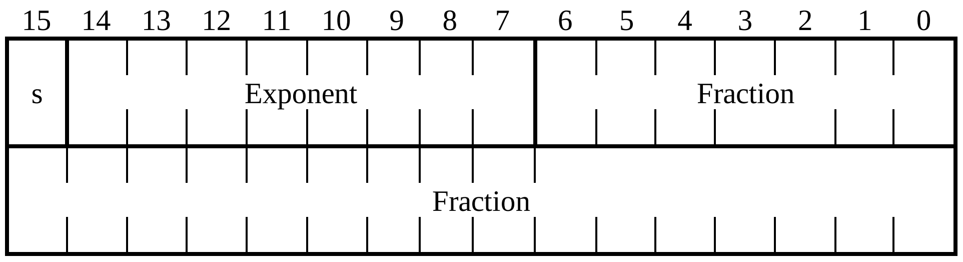
IEEE 32- AND 64-BIT FLOATING POINT FORMAT

In addition to the 48-bit format previously discussed, AMOS supports the IEEE format for 32-bit (single precision) and 64-bit (double precision) floating point numbers. These formats, because they have been standardized by the IEEE, are compatible across a wide variety of computer systems. This ensures that floating point data can be transferred between different system types, as well as ensuring a consistent level of accuracy across different computers.

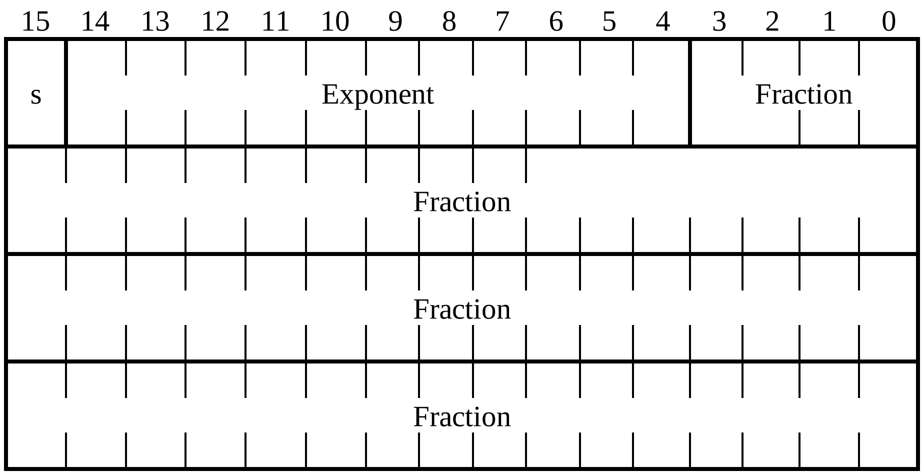
AMOS implements IEEE format floating point support in two ways: hardware and software. If a 68881 floating point coprocessor is present in your system, all floating point arithmetic is performed directly in hardware, giving the optimum in calculation performance. Because not all systems are equipped with a floating point coprocessor—some cannot physically accept the device—AMOS also provides a software emulation of the most common floating point operations.

What this means to you is complete transparency as to whether a system has hardware floating point or not. By limiting your use of 68881 instructions to the supported subset, your software runs regardless of the presence or lack of hardware floating point support.

The single-precision, 32-bit floating point format consists of a 23-bit mantissa, 8-bit biased exponent, and a sign-bit. The exponent is biased by 127. Single precision numbers are represented in two 16-bit words formatted as follows:



The double-precision, 64-bit floating point format consists of a 52-bit mantissa, 11-bit biased exponent, and a sign-bit. The exponent is biased by 1023. Double precision numbers are represented in four 16-bit words formatted as follows:



IEEE Format Floating Point Arithmetic

AMOS implements a subset of the 68881 floating point coprocessor instruction set in software. By limiting your use of 68881 instructions to the supported subset, your software runs regardless of the presence or lack of hardware floating point support. This section describes the subset implemented under AMOS.

If you choose to use 68881 instructions other than those described here, then your software will be limited to running only on those systems containing the coprocessor.

This section is not meant to be a complete reference to the 68881, but is instead intended to give you an overview of the supported instructions. For more complete information, please refer to the Motorola *MC68881 Floating Point Coprocessor User's Manual*.

Supported 68881 Floating Point Instructions

The following list describes the 68881 floating point coprocessor instructions that are supported by the AMOS software emulation capability. Also shown are the supported data types and the assembler syntax used with these instructions.

FABS Data format ***y***: B,W,L,S,D,X,P

```
FABSy   ea,FPn
FABSX   FPm,FPn
FABSX   FPn
```

FADD Data format ***y***: B,W,L,S,D,X,P

```
FADDy   ea,FPn
FADDX   FPm,FPn
```

FBCC Data format ***y***: W,L

```
FBCC    label
```

FCMP Data format ***y***: B,W,L,S,D,X,P

```
FCMPy   ea,FPn
FCMPX   FPm,FPn
```

FDIV Data format ***y***: B,W,L,S,D,X,P

```
FDIVy   ea,FPn
FDIVX   FPm,FPn
```

FINT Data format ***y***: B,W,L,S,D,X,P

```
FINTy   ea,FPn
FINTX   FPm,FPn
FINTX   FPn
```

FINTZ Data format ***y***: B,W,L,S,D,X,P

```
FINTZy  ea,FPn
FINTZX  FPm,FPn
FINTZX  FPn
```

FMOD Data format ***y***: B,W,L,S,D,X,P

```
FMODy   ea,FPn
FMODX   FPm,FPn
```

FMOVE Data format ***y***: B,W,L,S,D,X,P

```
FMOVEy  ea,FPn
FMOVEy  FPm,ea
FMOVEP  FPm,ea:#k ; k is k-factor
FMOVEP  FPm,ea:Dn ; Dn is dynamic k-factor
```

FMUL Data format ***y***: B,W,L,S,D,X,P

```
FMULy   ea,FPn
FMULX   FPm,FPn
```

FREM Data format ***y***: B,W,L,S,D,X,P

```
FREMy   ea,FPn
FREMX   FPm,FPn
```

FSQRT Data format ***y***: B,W,L,S,D,X,P

```
FSQRTy  ea,FPn
FSQRTX  FPm,FPn
FABSX   FPn
```

FSUB Data format ***y***: B,W,L,S,D,X,P

```
FSUBy   ea,FPn
FSUBX   FPm,FPn
```

IEEE Format Floating Point Conversion

AMOS supports calls to convert 48-bit floating point format numbers to and from IEEE format 32- and 64-bit floating point numbers. These calls allow you the freedom to mix floating point formats when dealing with data from multiple sources. Care must be exercised, however, to ensure that proper accuracy is maintained across the differing formats.

IEEE format floating point numbers may be converted to and from binary and packed decimal form via the 68881 floating point instruction FMOVE, discussed in the previous section.

FATOIS - Convert 48-bit Format to IEEE 32-bit Format

The FATOIS monitor call converts a 48-bit format floating point number to an IEEE format 32-bit single precision floating point number. The calling format is as follows:

```
FATOIS   adr
```

adr specifies the address from which to select the 48-bit floating point number as well as the address in which to store the converted result.

Note that only memory effective addresses are valid as operands.

FATOID - Convert 48-bit Format to IEEE 64-bit Format

The FATOID monitor call converts a 48-bit format floating point number to an IEEE format 64-bit double precision floating point number. The calling format is as follows:

```
FATOID   adr
```

adr specifies the address from which to select the 48-bit floating point number as well as the address in which to store the converted result. Be sure to allow sufficient space for the larger result.

Note that only memory effective addresses are valid as operands.

FISTOA - Convert IEEE 32-bit Format to 48-bit Format

The FISTOA monitor call converts an IEEE format 32-bit single precision floating point number to a 48-bit format floating point number. The calling format is as follows:

```
FISTOA   adr
```

adr specifies the address from which to select the 32-bit floating point number as well as the address in which to store the converted result. Be sure to allow sufficient space for the larger result.

Note that only memory effective addresses are valid as operands.

FIDTOA - Convert IEEE 64-bit Format to 48-bit Format

The FIDTOA monitor call converts an IEEE format 64-bit single precision floating point number to a 48-bit format floating point number. The calling format is as follows:

```
        FIDTOA   adr
```

adr specifies the address from which to select the 32-bit floating point number as well as the address in which to store the converted result.

Note that only memory effective addresses are valid as operands.

IEEE Format Floating Point Input/Output Calls

AMOS supports three IEEE format floating point input/output calls: one to get a floating point number from a memory buffer, one to get a floating point number from an IO device, and one to output floating point numbers. Each of these calls is provided in a single (32-bit) and double (64-bit) precision form.

GTFLTS - Get a Single Precision Floating Point Number

The GTFLTS monitor call converts an ASCII representation of a floating point number to 32-bit IEEE format single precision floating point format. GTFLTS uses A2 as a pointer to the ASCII input string, and stores the result in the specified destination. It updates A2 to point to the character following the floating point number. A floating point number is terminated by the first character that is not a legal continuation of the number.

The calling format is:

```
        GTFLTS   dst
```

Note that only memory effective addresses are valid as the destination operand.

The format of the ASCII input consists of an optional sign ("+" or "-"), up to twelve significant decimal digits with an optional embedded decimal point ("."), and an optional exponent. If an exponent is specified, it must consist of an "E" followed by an optional sign ("+" or "-") and the exponent value itself (in the range ±38). For example

1	3.14159265357	1E38
1.0	-1.123E-12	+12.4

You may use the OT$NLD flag with GTFLTS to disable language definition files, if the characters caused by the language interfere with the read. The format is:

```
        GTFLTS   @A5,OT$NLD
```

GTFLTD - Get a Double Precision Floating Point Number

The GTFLTD monitor call converts an ASCII representation of a floating point number to 64-bit IEEE format double precision floating point format. GTFLTD uses A2 as a pointer to the ASCII input string, and stores the result in the specified destination. It updates A2 to point to the character following the floating point number. A floating point number is terminated by the first character that is not a legal continuation of the number.

The calling format is:

```
GTFLTD  dst
```

Note that only memory effective addresses are valid as the destination operand.

The format of the ASCII input consists of an optional sign ("+" or "-"), up to twelve significant decimal digits with an optional embedded decimal point ("."), and an optional exponent. If an exponent is specified, it must consist of an "E" followed by an optional sign ("+" or "-") and the exponent value itself (in the range -308 to +307). For example:

1	3.14159265357	1E38
1.0	-1.123E-00	+12.4

You may use the OT$NLD flag with GTFLTD to disable language definition files, if the characters caused by the language interfere with the read. The format is:

```
GTFLTD  @A5,OT$NLD
```

GTFLFS - Get a Single Precision Floating Point Number from a File

The GTFLFS monitor call performs the same function as the GTFLTS call, except that it reads its input string from a file whose DDB is indexed by A2, rather than from a memory buffer.

The calling format (with A2 indexing a DDB), is as follows:

```
GTFLFS  dst
```

Note that only a memory effective address may be used as the destination operand.

You may use the OT$NLD flag with GTFLFS to disable language definition files, if the characters caused by the language interfere with the read. The format is:

```
GTFLFS  @A5,OT$NLD
```

GTFLFD - Get a Double Precision Floating Point Number from a File

The GTFLFD monitor call performs the same function as the GTFLTD call, except that it reads its input string from a file whose DDB is indexed by A2, rather than from a memory buffer.

The calling format (with A2 indexing a DDB), is as follows:

```
GTFLFD  dst
```

Note that only a memory effective address may be used as the destination operand.

You may use the OT$NLD flag with GTFLFD to disable language definition files, if the characters caused by the language interfere with the read. The format is:

```
        GTFLFD   @A5,OT$NLD
```

FCVTS, FCVTD - Output an IEEE Format Floating Point Number

The FCVTS and FCVTD monitor calls returns an ASCII representation of an IEEE format single or double precision floating point number to memory, to a file, or to the user's terminal. It outputs the number in the standard format shown below.

FCVTx is called in the following format:

```
        FCVTx    src,size,flags,precision,scale
```

x specifies whether a single or double precision source is specified.

The operand fields are defined as follows:

src	A memory effective address specifying where the floating point number to be displayed is to be fetched from.
size	Specifies the minimum width of the field to be displayed. The number appears right justified within the field, padded with leading spaces.
flags	Contains formatting flags defined as follows:

Symbol	Meaning
OT$FIX	Fixed format. "Precision" actually contains the number of digits to print after the decimal point.
OT$TRM	Output the result to the user's terminal.
OT$DDB	Output the result to the file whose DDB is indexed by A2.
OT$MEM	Output result to memory buffer indexed by A2 and update A2.
OT$LSP	Add one leading space to the result.
OT$TSP	Add one trailing space to the result.
OT$NSP	Don't replace the leading "-" with a space on positive numbers.
OT$NLD	Disable all use of language definition files.

precision	If bit 0 of flags is a zero, then precision specifies how many decimal digits to round to before displaying the number (the default is 15). If bit 0 of flags is a one, then precision specifies how many decimal digits to display after the decimal point (the default is zero).
scale	Specifies a decimal scaling factor. The floating point number is divided by the scaling factor prior to being output. The default scaling factor is zero.

IEEE Format Floating Point Error Trapping

During certain IEEE format floating point operations, error conditions can occur. These error conditions are: division by zero, numeric overflow, and numeric underflow. Additionally, if the floating point is being performed by hardware (as opposed to software simulation), the errors "branch or set on unordered condition," "signaling NAN," "operand error," and "inexact result" can occur. AMOS provides a method of trapping these errors so that your program can take the appropriate action.

If you do not specify that these errors are to be trapped, AMOS executes its default error processing routines which display an appropriate error message on your terminal and abort the program back to AMOS command level.

If you want instead to handle these errors yourself, you should place the address of your error processing routine in JOBIEE in your job's JCB. Then, when an error occurs, AMOS executes your error processing routine. Further information on JCBs and how to access them may be found in Chapter 2.

When your error processing routine is called, the cause of the error can be determined by examining register D0:

Octal Value	Hex Value	Meaning
1	1	Inexact result
4		Divide by zero was attempted
10	8	Floating underflow has occurred
20	10	Floating overflow has occurred
40	20	Operand error
100	40	Signaling NAN
200	100	Branch or set on unordered condition

If your trap routine simply executes a RTN instruction, default values are used as the result. For underflow, a value of 0.0 is supplied; for overflow the largest positive number is used; divide by zero and other errors are treated as fatal errors, causing the job to abort to AMOS command level.

Chapter 12
Generalized Output Monitor Calls

AMOS provides a series of monitor calls to use for generalized, destination independent output. These calls all duplicate functions you can perform with other calls, but are presented here in their general purpose form. These calls are very useful when doing output you may want to send to either a file or to the user's terminal. An example of such an application is the DSKANA program, which, at the user's option, sends its output to either the terminal or to a listing file.

Because the calls described in this chapter are generalized versions of other calls, they are inherently less efficient than the specific call. For example, the OUT call, when used to place a byte in a memory buffer can be replaced with a simple MOVB instruction that may be 30-40 times more efficient. For this reason, you should use these calls only when you do not have a fixed output destination.

OUTPUT FLAGS

All calls described in this chapter take a common set of flags that effect the destination of the output and specify its formatting. These flags are:

Symbol	Octal Value	Hex Value	Meaning
OT$TRM	2	2	Outputs the result to the user terminal
OT$DDB	4	4	Outputs the result to the file whose DDB is indexed by A2
OT$MEM	10	8	Puts the result in memory at the buffer indexed by A2 and updates A2

If the call does not specify any flags, the call fetches them from D6. Thus, the following two calls are equivalent:

```
OUT      OT$TRM
```

is the same as:

```
MOV      #OT$TRM,D6
OUT
```

OUT - OUTPUT ONE CHARACTER

The OUT call outputs one character from D1 to the destination specified by the formatting flags.

The calling sequence is:

```
MOVB     char, D1       ; get character to output
OUT      flags          ; output the character
```

OUTI - OUTPUT A STRING OF CHARACTERS

The OUTI call outputs a string of characters that 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 terminal:

```
OUTI     OT$TRM
ASCII    /line 1 data/
BYTE     15
ASCII    /line 2 data/
BYTE     15, 0
EVEN
```

The OUTI call automatically appends a line-feed to all carriage-returns included in the string.

OUTL - OUTPUT A STRING OF CHARACTERS INDEXED

The OUTL call is similar to the OUTI call in that it outputs a string of characters up to a null byte. But instead of following the call, the string of characters for the OUTL call may be located anywhere in memory. OUTL takes two arguments, a memory effective address specifying where the string is located, and the formatting flags. The OUTL call appends a line-feed to each carriage-return contained in the output string.

The calling sequence is:

```
MOV      ptr to string,adr      ; get pointer to string to output
OUTL     adr,flags              ; output the string
```

MESSAGE OUTPUT CALLS

Three calls are defined in SYS.UNV as macros using the OUTI call. These calls are for convenience and make your program easier to understand. They all take two arguments: the formatting flags and an ASCII message string that is to be output. Due to the way macro arguments are processed, if the message has leading or trailing spaces, or if it has embedded commas, it must be enclosed in angle brackets or part of it will be lost. The three calls are:

```
OUTS     flags,msg     ; outputs message as is
OUTSP    flags,msg     ; outputs message and appends one space
OUTCR    flags,msg     ; outputs message and appends a CRLF pair
```

SMSG - OUTPUT A SYSTEM MESSAGE

Outputs a system message from the SYSMSG message definition file to a terminal, memory buffer, or an output file (by using a DDB). The format is:

```
SMSG     msg #, flags
```

msg #	Is the message number to be output
flags	Specifies the destination and format of the ASCII text to be output:

Symbol	Octal Value	Hex Value	Meaning
OT$LDQ	1	1	Output a leading question mark before the message
OT$TRM	2	2	Output the specification to the user's terminal
OT$DDB	4	4	Output the specification to the DDB indexed by A2
OT$MEM	10	8	Output the specification to memory, using the pointer contained in A2
OT$LSP	20	A	Output a leading " - " before the message
OT$TSP	40	20	Output a trailing " - " after the message
OT$LPC	200	80	Output a leading percent sign before the message
OT$SPC	400	100	Output a leading space before the message
OT$CR	1000	200	Output a carriage-return line-feed pair after the message
OT$UPC	2000	400	Force the first character of the message to be upper case
OT$NUL	4000	800	Output a null after the message

For example, to output message #43 to the memory buffer BUFF(A5) with a leading question mark:

```
LEA      A2,BUFF(A5)
SMSG     #43.,OT$MEM!OT$LDQ
```

Chapter 13
Miscellaneous Monitor Calls

The monitor calls discussed in this chapter do not fit into any of the categories treated thus far.

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, deletes all temporary memory modules in the user partition and resets any parameters that are program dependent such as JOBBPT, etc. It also releases all assigned devices at this time, and returns the user terminal to the monitor command mode, ready to process another operator command.

Calling sequence:

```
        EXIT                ; back to AMOS
```

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 the monitor sets a flag in the JCB status word, which the program must test. An application program can use the CTRLC call 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 application program could perform the same task, with a few instructions, by locating its own JCB status word and checking the J.CCC flag within it. The format of this call is:

```
        CTRLC   routine-address
```

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 application 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.

JLOCK AND JUNLOK - PREVENT CONTEXT SWITCHING

The JLOCK call prevents context switches from occurring and allows the current user to run until a JUNLOK is performed. You can use this call to protect critical sections of code from being executed by more than one user at a time. A typical example of this would be a routine that sends a command byte to an I/O device controller and then reads the result. If no JLOCK was used, one user could send a command byte and then, before that user can read the result, a context switch to another job could occur. This new job could then execute the same routine, destroying the first job's result. By placing a JLOCK before the command write, and a JUNLOK after the result is read, you can avoid this problem.

You should only use JLOCKs in situations where they are truly necessary. Excessive or unwise use of JLOCKs can slow down the performance of your system and/or cause a deadlock situation. In almost all cases, use of RQST and RLSE is a better way of protecting critical sections of code. Also, don't use JLOCK and then perform I/O within the locked state—use JLOCKI (discussed below) in that case.

Calling sequence:

```
JLOCK                 ; prevent context switching
…                     ; code to be protected
JUNLOK                ; re-enable context switching
```

JLOCKI - WAIT FOR I/O, THEN PREVENT CONTEXT SWITCHING

The JLOCKI call is similar to the JLOCK call (discussed above) except that it waits for all I/O to be completed before granting the lock. Like JLOCK, JLOCKI prevents context switches from occurring and allows the current user to run until a JUNLOK is performed. You can use this call to protect critical sections of code from being executed by more than one user at a time.

JLOCKI is used exactly like JLOCK in situations where disk I/O is to be performed within the locked state. It must not be used within drivers or anywhere else where the caller is already inside an I/O operation, or a deadlock will result.

The same warnings concerning excessive use of the JLOCK call (discussed above) also apply to JLOCKI.

Calling sequence:

```
JLOCKI                ; wait for I/O before preventing context
…                     ; switching code to be protected
JUNLOK                ; re-enable context switching
```

PLOCK AND PUNLOK - PREVENT PROCESS CONTEXT SWITCHING

The PLOCK and PUNLOK calls work very much like the JLOCK and JUNLOK calls, except that instead of locking a single job, they lock the issuing job and any processes it may have spawned.

These calls are particularly useful in multi-tasking application software which must allow all spawned processes to run, but does not wish other jobs or processes to be able to run.

The first PLOCK call issued by a job prevents any process not related to the issuing job from running. "Related" means that it is either a parent or child process. The relation can go up or down as many levels as currently exist.

PLOCK calls may be nested, and multiple PLOCK calls issued by different but related processes are properly handled.

Calling sequence:

```
PLOCK                 ; prevent context switching
```

```
              …                      ; code to be protected
              PUNLOK                 ; re-enable context switching
```

RSTCON - RESTORE CONTEXT

The RSTCON call resets the stack pointer (SP) to the base of the stack within the current context. RSTCON is a convenient way to reset the stack pointer without interfering with the operation of the AMOS or PCALL monitor calls.

The calling format is:

```
              RSTCON                 ; reset stack point to base of stack
```

RQST - REQUEST CONTROL OF A SEMAPHORE

A0 points to a 4-word semaphore which may conventionally be associated with any type of resource (disk, buffer, queue block, etc.). When a job requires access to a resource, it should RQST the semaphore associated with that resource. RQST decrements the semaphore count (representing the number of available resources) by 1. If the resulting count is greater than or equal to 0, RQST returns and allows access to that resource. If the difference is less than 0, RQST places the job in a wait chain until the resource is available.

To illustrate: suppose a job needs to access one of 20 available queue blocks. A semaphore with an initial value of 20 (to represent the available queue blocks) could be set up and accessed prior to any attempts to allocate a queue block. A RQST call decrements the count from 20 to 19, confirms that 19 is greater than or equal to 0, then returns control of the job so it can get a queue block. If none of the 20 queue blocks were available (i.e., the semaphore count ≤ 0), the job would be placed in a wait state until a queue block was identified as freed via a RLSE call. (See Section 13.8.)

Sample calling sequence (based on example above):

```
              LEA     A0,SEM         ; index the semaphore
              RQST                   ; request the semaphore
              ...

      SEM:    LWORD    20.           ; count of available blocks
              LWORD    0             ; place for wait queue
```

For additional information on semaphores and their use, consult any one of the many available operating systems textbooks.

RLSE - RELEASE CONTROL OF A SEMAPHORE

If, upon execution of the RQST call (see the previous section), the semaphore count is less than or equal to 0 (i.e., none of the resources requested is available), RQST puts the requesting job to sleep in a wait chain. When one job is finished with one of those resources, a RLSE call on the semaphore associated with that resource increments the count by 1 and determines if the result is less than or equal to 0. If it is, RLSE awakens the next job in the wait chain and allows it to finish the RQST.

For example, if none of 20 queue blocks is currently available, the count is less than or equal to 0—let's say it's 0. Before a job tries to get a queue block, a RQST on the semaphore decrements the count from 0 to -1 and places the job in a wait chain. After a job frees a queue block, it uses the RLSE call on the semaphore associated with "queue blocks." This call increments the semaphore count by 1, resulting in 0, and wakes the first job in the wait chain, which allows it to continue and allocate a queue block. The following diagram illustrates the semaphore:

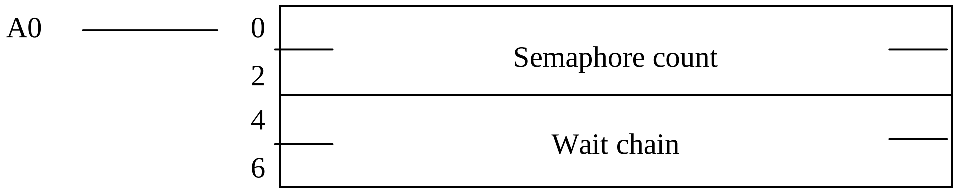


Figure 13-1: Semaphore Layout

Sample code to release the semaphore we requested in the RQST example above is as follows:

```
LEA      A0, SEM       ; index the semaphore
RLSE                   ; release it
```

PCALL - INVOKE PROGRAM AS SUBROUTINE

PCALL is similar to the standard machine instruction call (CALL), except return is not done via the RTN instruction but is accomplished via the EXIT supervisor call. The format is:

```
PCALL    subroutine-address
```

subroutine-address is the address of the program you wish to call.

AMOS - EXECUTE AMOS COMMAND AS SUBROUTINE

When AMOS is used as a monitor call, it treats the character string pointed to by A2 as a monitor command line, and it executes the AMOS command in this command line without leaving the current program. The calling sequence is:

```
MOV      ptr to string, A2      ; index the command string
AMOS
```

Be sure to terminate the command string properly. Most AMOS programs expect the command string to end with a CR/LF pair.

Although the AMOS call was intended for use with programs, it can also be used to execute command files. Since the monitor handles command files differently than programs, the AMOS call acts a little differently for command files than for programs:

1. Each time the AMOS call is used in your program to invoke a command file, that command file is queued in memory *without being executed.*
2. When your program exits, the monitor executes the command files in reverse of the order in which they were queued in memory.

TIMER - ENTER ITEM INTO TIMER QUEUE

The operating system maintains an internal clock-driven queue. The monitor processes the items in this queue after a specific amount of time has elapsed. The TIMER monitor call allows you to insert items into this queue. The call format is:

```
        TIMER   adr          ; insert item into queue
```

The specified argument must point to a timer block (typically a monitor queue block) formatted as follows:

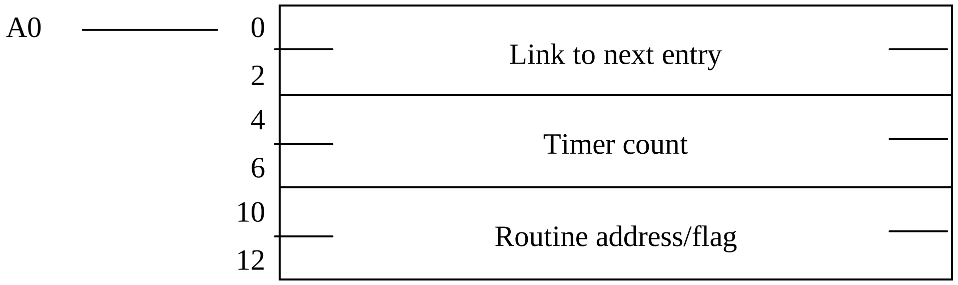


Figure 13-2: Timer Block

Where the "Link to next entry" is set by the TIMER call, "Timer Count" is the unsigned number of clock ticks you wish to wait before the queue entry is processed, and "Routine Address/Flag" contains one of two items:

1. If "Routine Address/Flag" is non-zero, it is treated as the address of a routine to be called when the specified amount of time has elapsed.
2. If "Routine Address/Flag" is zero, then it will be set non-zero when the specified amount of time has elapsed.

If you specify that a routine is to be executed, that routine may modify registers A0, A6, D0, D6, and D7. All other registers must be preserved. Remember that this routine is being called at the interrupt level; therefore, you should take extreme care to minimize the amount of time spent in the routine.

The timer queue is driven by the system real-time clock, which operates with a precision of 100 microseconds.

When your timer routine is executed, register A0 will be indexing the word immediately following the routine address/flag word in the timer queue entry. By subtracting 14_8 from this register, it can be made to point to the first word of the timer queue entry, easing the task of re-inserting the timer block via another TIMER call, or returning the queue block via a QRET call.

You may execute a TIMER call from within a routine that is called via the TIMER call to cause repetitive calls to your timer routine.

Once the specified amount of time has elapsed, and the routine has been called or the flag set, the timer queue entry is de-queued. If you wish to have a routine called repeatedly, you must re-insert the timer queue entry via another TIMER call.

If a routine is initiated with the TIMER call, then that routine cannot use the SLEEP call.

To remove a timer queue entry that has not yet expired, use the DQTIMR call, described below. You must use a system queue block as the timer block if you intend to use DQTIMR.

DQTIMR - REMOVE ITEM FROM TIMER QUEUE

When using the timer queue, situations arise where an active timer queue element must be removed from the queue before it has timed out. The DQTIMR routine provides a standard method for removing such active elements. The call format for DQTIMR is:

```
        DQTIMR   adr          ; remove item from queue
```

The specified argument must point to an active timer block that is currently within the system timer queue. This entry will be removed from the timer queue, and any other entries in the queue will be adjusted as necessary.

Note that this call only removes the timer block from the active timer queue. This call assumes the timer queue entry being de-queued is a system queue block—it won't work if it isn't.

If the address you specify is not an active timer queue block, this call does nothing. To avoid possible deadlocks, your program should be SVLOKed, or otherwise protected, to prevent a timer block from expiring prior to attempting to de-queue it.

ERRMSG - OUTPUT STANDARD ERROR MESSAGE

The ERRMSG call provides a standard method of outputting system error messages. ERRMSG accepts two arguments: the number of the standard system error to be displayed, and flags that specify the output destination and formatting. You can find a list of the standard system errors that can be displayed with ERRMSG under Error Codes in Chapter 6.

The flags used with ERRMSG are:

Symbol	Octal Value	Hex Value	Meaning
OT$LDQ	1	1	Outputs a leading question mark ("?") and capitalizes the first character of the error message
OT$TRM	2	2	Outputs the result to the user terminal
OT$DDB	4	4	Outputs the results to the file whose DDB is indexed by A2
OT$MEM	10	8	Puts the result in memory at the buffer indexed by A2 and updates A2
OT$LSP	20	10	Adds a " - " sequence to the start of the message
OT$TSP	40	20	Adds a " - " at the end of the message

The calling sequence is:

```
        ERRMSG   src, flags
```

For example, if A3 points to a DDB which contains an error code, and you want to display that error on the user's terminal, with a leading question mark, you would specify:

```
ERRMSG   D.ERR(A3), OT$TRM!OT$LDQ
```

If the error byte (D.ERR) of the DDB contained a 3 (D$EFNF) error code, the example above would display "?File not found" on the user's terminal.

SMSG - OUTPUT SYSTEM MESSAGE

The SMSG call provides a method of outputting standard system messages from the system message file SYSMSG. By using this single, centralized message file, messages can easily be translated into different languages. Along with the language definition tables, this facility allows a single system to support multiple languages at the same time.

SMSG accepts two arguments: the number of the system message to be displayed and flags that specify the output destination and formatting.

Note that SMSG destroys the contents of register D1, overwriting it with the message number you specify.

The flags used with SMSG are:

Symbol	Octal Value	Hex Value	Meaning
OT$LDQ	1	1	Outputs a leading question mark ("?") and capitalizes the first character of the error message
OT$TRM	2	2	Outputs the result to the user terminal
OT$DDB	4	4	Outputs the results to the file whose DDB is indexed by A2
OT$MEM	10	8	Puts the result in memory at the buffer indexed by A2 and updates A2
OT$LSP	20	10	Adds a " - " sequence to the start of the message
OT$TSP	40	20	Adds a " - " at the end of the message
OT$LPC	200	80	Precedes the message with a "%"
OT$SPC	400	100	Adds a space to the end of the message
OT$CR	1000	200	Adds a carriage return to the end of the message
OT$UPC	2000	400	Forces the first character of the message to upper case
OT$NUL	4000	800	Adds a null byte to the end of the message

The calling sequence is:

```
SMSG    message, flags
```

For example, if you want to output message number 5 on the user's terminal, with a leading question mark, you would specify:

```
SMSG    #5, OT$TRM!OT$LDQ
```

STDERR - PERFORM STANDARD ERROR PROCESSING

When a program is processing its own exceptions via the JOBERC vector, there are times when it is best to allow AMOS to perform its own normal exception processing. A typical example would be where a program wanted to handle one specific type of error, but wanted all others to perform normal exception processing.

The STDERR monitor call provides a method by which an exception that has caused a trap to a job's JOBERC vector can abort the JOBERC routine and resume normal exception processing. This call can only be executed from within a JOBERC exception routine.

The calling format is:

```
              STDERR
```

This call takes no arguments and does not return to the calling program.

SVLOK - DISABLE INTERRUPTS

The SVLOK monitor call enables certain critical operations to execute as one single unit, without the possibility of interrupts. SVLOK disables all interrupts until the processor is "unlocked" via an SVUNLK monitor call.

The calling format is:

```
              SVLOK                ; lock out interrupts
```

Note that SVLOK cannot be nested; that is, no matter how many SVLOK instructions are executed, one single SVUNLK instruction re-enables interrupts.

The SVLOK call can only be used while in supervisor mode. Any attempt to use SVLOK when in user mode will result in a "privileged instruction" error trap.

The SVLOK call is intended for use only by operating system routines. Improper use of this monitor call can cause system failure.

Future versions of AMOS may not support the use of the SVLOK call. For this reason, we recommend you don't use it.

SVUNLK - ENABLE INTERRUPTS

The SVUNLK monitor call enables processor interrupts that have been disabled by the SVLOK monitor call.

The calling sequence is:

```
              SVUNLK               ; enable interrupts
```

The SVUNLK call can only be used while in supervisor mode. Any attempt to use SVUNLK when in user mode will result in a "privileged instruction" error trap.

The SVUNLK call is intended for use only by operating system routines. Improper use of this monitor call can cause system failure.

Future versions of AMOS may not support the use of the SVUNLK call. For this reason, we recommend that you don't use it.

SUPVR - ENTER SUPERVISOR MODE

The 680xx family of processors supports at least two distinct processor modes: Supervisor and User. Certain machine instructions are not allowed in user mode. The SUPVR monitor call allows a user to enter supervisor mode to have access to these instructions.

The calling sequence is:

```
        SUPVR             ; enter supervisor state
```

The user may re-enter user mode by use of the LSTS instruction.

The SUPVR call is intended for use only by operating system routines. Improper use of this monitor call can cause system failure.

Future versions of AMOS may not support the use of the SUPVR call. For this reason, we recommend that you don't use it.

DEVCHR - DETERMINE DEVICE CHARACTERISTICS

The DEVCHR calls returns characteristics of a given device, allowing the program to tailor its operation based on the type of device in use. Gives you a brief description of the specified device in a standard form, so you do not have to search tables and drivers. The format is:

```
        DEVCHR  ddb, table
```

ddb points to a standard DDB containing the specification of the device you want information on. This DDB must be INITed, and have a buffer allocated for it prior to the DEVCHR call. The parameter ***table*** is a pointer to a table where the information will be placed by the call. The argument block pointed to by "table" is formatted as:

Symbol	Meaning
DC.FLG	A 32-bit field containing flags that describe device features. The flags are: DC$FSD Device is file structured DC$ATT Device uses an alternate track table to track bad areas DC$LOG Device is a logical unit DC$SHR Device is sharable DC$ASN Device is currently assigned to a job DC$MNT Device is currently mounted DC$NAC Device is set to "no network access" DC$14D Device uses extended format directories
DC.BLK	32-bit field containing the number of blocks on the device
DC.ATT	32-bit field containing the number of bytes in the alternate track table associated with the device. Unless DC$ATT flag is set, this entry is meaningless
DC.BUF	32-bit field containing the standard buffer size used by the device, expressed in bytes
DC.BMS	32-bit field containing the size of the bitmap used by the device. Meaningless unless DC$FSD flag is set.
DC.JOB	32-bit field containing a pointer to the JCB of the job the device is assigned to. Only valid when the device is NOT sharable (DC$SHR = 0) and the device is assigned DC$ASN = 1)
DC.PHY	16-bit field containing the physical device number of the device
DC.LOG	16-bit field containing the logical unit number, within the physical unit, of the device
DC.SPR	20 bytes reserved for future expansion

To make it easy to allocate the space required to hold the returned arguments, the symbol DC.SIZ defines the number of bytes returned by DEVCHR.

Here is a sample use of DEVCHR:

```
FSPEC    DDB(A5)          ; get the device name
INIT     DDB(A5)          ; INIT a buffer
DEVCHR   DDB(A5),DC(A5)   ; get the characteristics
MOV      DC+DC.BUF(A5),D1 ; get buffer size
```

DSKFRE - DETERMINE NUMBER OF FREE DISK BLOCKS

Returns the number of free blocks contained on a file-structured device, as well as the largest contiguous free space contained on the device. The format is:

```
DSKFRE   ddb, table
```

ddb points to a standard DDB containing the specification of the device you want information on. This DDB must be INITed, and have a buffer allocated for it prior to the DSKFRE call.

The parameter ***table*** is a pointer to a 24-byte table where the information will be placed by the call. The argument block pointed to by table is formatted as:

Symbol	Meaning
DF.FRE	32-bit field containing the total number of available blocks.
DF.HOL	32-bit field containing the size of the largest group of contiguous free blocks.
DF.SPR	Reserved for future expansion.

To make it easy to allocate the space required to hold the returned arguments, the symbol DF.SIZ defines the number of bytes returned by DSKFRE.

The following code fragment demonstrates the use of DSKFRE:

```
FSPEC    DDB(A5)          ; get the device name
INIT     DDB(A5)          ; INIT a buffer
DSKFRE   DDB(A5),DF(A5)   ; get the characteristics
MOV      DF+DF.FRE(A5),D1 ; get number of free blocks
```

TDVCNG - CHANGE TERMINAL DRIVERS

The TDVCNG monitor call allows you to change the terminal driver associated with a given terminal control block. This is particularly useful when a terminal control block may be controlling a dialup connection, where the type of terminal to be used is not known in advance.

The TDVCNG call is called with A2 indexing an ASCII string specifying the name of the terminal driver to be selected. The calling sequence is:

```
TDVCNG   flags{, tcb}
```

flags is a register containing flags which specify actions to be taken when the terminal driver change is made. After completion of the TDVCNG call, a result code is returned in this register.

The optional ***tcb*** argument allows you to specify the TCB address of the terminal whose terminal driver you wish to change. If this argument is omitted, the terminal attached to the current job will be changed.

The only flag that can be specified is:

Symbol	Value	Meaning
TN$INI	1	If set, the terminal driver initialization routine will be called after the change is made

The result codes returned in the flags register are:

Symbol	Value	Meaning
TN$TNF	1	The specified terminal driver was not found
TN$IMP	2	The available terminal driver impure area is not large enough to support the new terminal driver

The Z-bit will be set upon return from the TDVCNG call if the change was made successfully.

In order to be selected as a new terminal driver, the terminal driver must have been loaded into the terminal driver chain by TRMDEF or TDVDEF in the system initialization command file. You can determine which terminal drivers are available by using the TDVDEF command after system startup.

Because some terminal drivers require differing amounts of impure space, it is not always possible to change from a terminal driver that uses a small amount of space to one that uses a large amount. Normally this is not a problem since most terminal drivers use only a small impure area. If a situation arises that presents a problem with impure space allocation, change your system initialization file to specify the terminal driver with the large impure requirement within the TRMDEF command, thus reserving the larger amount of space required. You can then change terminal drivers (via the SET TDV command) to the driver you actually wish to use.

The terminal driver initialization routine should normally be called whenever the terminal driver is changed. This ensures that the physical terminal and the new terminal driver agree with each other, as to currently selected operational modes and settings. However, the terminal driver initialization routine must perform some output to the screen, potentially causing distracting "flashing" on the terminal. For those applications where this is not acceptable, you can suppress the calling of the initialization routine by not including the TN$INI flag.

The following code fragment changes the current job's terminal driver to AM60.TDV, and requests that the terminal driver initialization be performed.

```
        LEA     A2, TDVNAM    ; index the new driver name
        MOV     #TN$INI,D0    ; select initialization
        TDVCNG  D0            ; change terminal drivers
        BEQ     ALLOK         ;   change successful -
        CMP     D0,#TN$INI    ; driver not found?
        BEQ     NOTFND        ;   yes -
        CMP     D0,#TN$IMP    ; not enough impure space?
        BEQ     NOIMP         ;   yes -
        TYPECR  <?Unknown error>
        EXIT                  ; return to AMOS

TDVNAM: ASCIZ   /AM65AX/      ; new driver name
        EVEN
```

LEVEL7 - TRANSFER CONTROL TO LEVEL7 DEBUGGER

The LEVEL7 monitor call lets you debug a problem more effectively by using the Level7 debugger. If you encounter a problem which you cannot solve using the system debugger (FIX) or debugging code in your program, you may want to use LEVEL7.

LEVEL7 locks the system and places you into the Level7 debugger. While in Level7, you can investigate the system resources. Once you exit Level7, the system returns to its prior state and most other processes resume running. In some cases, not all other processes may resume; if this happens, you will need to reboot your computer.

This monitor call should be used only by experienced programmers who are investigating a problem which they cannot fix any other way. If it is not used properly, it could cause system damage!

Do not use this monitor call unless you are totally familiar with the system debugger (FIX).

The calling sequence is:

```
        LEVEL7              ; Enter Level7 processing
```

For information about using the Level7 debugger, please see the *Level7 User's Guide*.

ICOFF - TURN INSTRUCTION CACHE OFF

The ICOFF call checks to see what type of processor it is executing on, then performs the proper instructions to turn the instruction cache off. The current cache lines are invalidated before the cache is turned off.

Calling sequence is:

```
        ICOFF
```

If you do need to use this call to turn off instruction cache for some reason, be sure to turn the cache back on when you are done. Leaving the instruction cache off can hurt system performance.

ICON - TURN INSTRUCTION CACHE ON

The ICON call checks to see what type of processor it is executing on, then performs the proper instructions to turn the instruction cache on. The current cache lines are invalidated before the cache is turned on.

Calling sequence is:

```
        ICON
```

SYNC - FLUSH WRITE CACHE BLOCKS

The SYNC call has been added to AMOS 2.3A(485) and later to allow programs to flush the write cache on demand. The monitor call will allow for specific devices (dsk, sub, …) or all devices to be flushed. In all cases, all logicals for the specified device will be flushed.

Calling sequence is:

```
        CLR     D6          ; flush all write caches
        SYNC                ; flush write cache blocks

    or

        MOVW    #[DSK],D6   ; flush only dsk device
        SYNC                ; flush write cache blocks
```

Use this call prudently, since it can slow the system down and diminish the usefulness of write caching.

SPAWN - SPAWN A NEW JOB

The SPAWN call will allow programs to spawn jobs to perform specific tasks. This will free up the main job from tasks that can be done in the background. This is seen in the new Task Manager (TASKIT.LIT) which spawns a job for each printer and batch processing job defined. These SPAWNed jobs do only one thing - printing files to a specific printer or processing batch request.

The SPAWN monitor call uses one argument, an SCB (Spawn Control Block), which gives the monitor call all the information it needs to create and support the SPAWNed job. It is the programmer’s responsibility to clear and fill the SCB with appropriate data. The calling sequence is:

```
        SPAWN   SCB(A5)       ; spawn a job
        BNE     ERR01         ; an error has occurred
        MOV     A0,JCB1(A5)   ; save spawned job’s JCB
        MOV     A1,TCB1(A5)   ; save spawned job’s TCB
        .
ERR01:  .                     ; what type of error is it?
```

The SCB is as follows:

```
SP.FLG:     BLKL   1                     ; Spawn Flags
SP.RSIZ:    BLKL   1                     ; Requested Memory Size
SP.NJCB:    BLKL   1                     ; JCB Name (Create)      (RAD50)
SP.NTCB:    BLKL   1                     ; TCB Name (Create/Use)  (RAD50)
SP.NTDV:    BLKL   1 ; NOT IMPLEMENTED;    TCB's TDV Name to use  (RAD50)
SP.NIDV:    BLKL   1 ; NOT IMPLEMENTED;    TCB's IDV Name to use  (RAD50)
SP.NSM:     BLKL   1                     ; SMEM's name to use     (RAD50)
SP.CMD:     BLKL   1                     ; Program Start/Command Line Index
SP.IBS:     BLKL   1                     ; TCB's Input Buffer Size
SP.LBS:     BLKL   1                     ; TCB's Line Buffer Size
SP.OBS:     BLKL   1                     ; TCB's output Buffer Size
SP.SA0:     BLKL   1                     ; A0 contents on execution
SP.SA1:     BLKL   1                     ; A1 contents on execution
SP.SA2:     BLKL   1                     ; A2 contents on execution
SP.SA3:     BLKL   1                     ; A3 contents on execution
SP.SA4:     BLKL   1                     ; A4 contents on execution
SP.SA5:     BLKL   1                     ; A5 contents on execution
SP.SA6:     BLKL   1                     ; A6 contents on execution
SP.SD0:     BLKL   1                     ; D0 contents on execution
SP.SD1:     BLKL   1                     ; D1 contents on execution
SP.SD2:     BLKL   1                     ; D2 contents on execution
SP.SD3:     BLKL   1                     ; D3 contents on execution
SP.SD4:     BLKL   1                     ; D4 contents on execution
SP.SD5:     BLKL   1                     ; D5 contents on execution
SP.SD6:     BLKL   1                     ; D6 contents on execution
SP.SD7:     BLKL   1                     ; D7 contents on execution
SP.JCB:     BLKL   1                     ; SPAWNed job's JCB index
SP.MIDX:    BLKL   1                     ; Memory Index to use for SPAWNing
SP.PRI:     BLKB   1                     ; Job's Priority to assign
SP.SP2:     BLKB   3                     ; Spare 3 Bytes not currently used!
SP.SP1:     BLKL   3                     ; Spares not currently used!
SP.SP0:     BLKL   10.                   ; Spares not currently used!
SP.SIZ:                                  ; SPAWN Control Block (SCB) size
```

The SPAWN flags (SP.FLG) are as follows:

```
SPN.JCB    = 0001                        ; Use JCB's memory
 SPN%JCB   = 0.                          ; BTST SYMBOL
SPN.SM     = 0002                        ; Use SMEM for memory
 SPN%SM    = 1.                          ; BTST SYMBOL
SPN.BU     = 0004                        ; Use BOOT up memory
 SPN%BU    = 2.                          ; BTST SYMBOL
SPN.NTCB   = 0010                        ; Use/Create TCB with Name in SP.NTCB
 SPN%NTCB  = 3.                          ; BTST SYMBOL
SPN.RUN    = 0020                        ; Run SPAWNed job at SP.CMD index
 SPN%RUN   = 4.                          ; BTST SYMBOL
SPN.CMD    = 0040                        ; Run SPAWNed job at SP.CMD index
 SPN%CMD   = 5.                          ; BTST SYMBOL
SPN.ALL    = 0100                        ; Kill all jobs SPAWNed by this job
 SPN%ALL   = 6.                          ; BTST SYMBOL
SPN.NJCB   = 0200                        ; Create JCB with Name in SP.NJCB
 SPN%NJCB  = 7.                          ; BTST SYMBOL
SPN.ITCB   = 0400                        ; TCB index exist
 SPN%ITCB  = 8.                          ; BTST SYMBOL
SPN.MIDX   = 1000                        ; Use the memory indexed by SP.MIDX
 SPN%MIDX  = 9.                          ; BTST SYMBOL
SPN.PERM   = 2000                        ; Permanent SPAWN Job Flag
 SPN%PERM  = 10.                         ; BTST SYMBOL
SPN.B11    = 4000                        ; Not used
 SPN%B11   = 11.                         ; BTST SYMBOL
SPN.B31    = 20000000000                 ; Not used
 SPN%B31   = 31.                         ; BTST SYMBOL
```

The Error Codes are as follows:

00	SPAWN request was successful
01	Job table is full
02	Unable to locate PSEUDO.IDV
03	Unable to locate PSEUDO.TDV
04	Invalid parameter requested
05	Not used
06	Invalid parameter requested
07	Insufficient memory for request
08	TCB requested is attached to a job
09	System is up and running
10	Must be 68020 or above
11	JCB name exist
12	Invalid SMEM name requested
13	No requested memory size
14	Provided memory index is zero
15	TCB requested not found
16	SMEM requested, but SMEM not defined

When the SPAWN Monitor call is used, AMOS will create the JCB, TCB, get the memory from the requested area, and return a completion status code in D0. The Z-flag will be set if successful and reset if not. The SPAWNed job will be put into the run queue to start executing at the spot requested. The address register and data registers will be set to the values requested in the SCB and the job will continue to run until it performs an EXIT call or the Parent job performs an EXIT.

Not all combinations have been tested, but the majority of the standard uses have been.

If you create and delete many SPAWNed jobs in a program and the memory is acquired from the job's partition, you may run into fragmentation problems or other catastrophic problems if care is not taken when using this monitor call.

GETVTI - IDENTIFY VIRTUAL TERMINAL SOURCE

The GETVTI call lets you identify the client when running under a virtual terminal session. This works for both AlphaTCP and AlphaNET.

The call takes four arguments as follows:

```
        GETVTI tcb, host, instance, protocol
```

The ***host***, ***instance***, and ***protocol*** arguments should point to areas capable of holding up to 100 characters. They will be filled in with null terminated ASCII strings.

host receives the remote host identifier: the IP address or the AlphaNET CPU ID of the connecting system.

instance receives a connection identifier: the remote port number or AlphaNET socket number.

protocol receives a connection type identifier: "TCP/IP" or "ANET"

The GETVTI call requires AMOS 2.3A or later. To determine if AMOS supports GETVTI, check the SY1$DYNSZ flag in the SYSTEM1 longword in the AMOS communication area.

Here is an example, assumes A4 indexes impure memory:

```
MOV     SYSTEM1,D7              ; get secondary system flags
AND     #SY1$DYNSZ,D7           ; support dynamic size tcbs?
BEQ     NOTSUP                  ;    no - then no GETVTI either
JOBIDX                          index this job
MOV     JOBTRM(A6),D7           ; have terminal?
BEQ     NOTERM                  ;    no -
MOV     D7,A5
GETVTI  @A5,HOST(A4),INST(A4),PROT(A4)
BMI     NOTEXT                  ; not extended tcb
BCS     NOTVTM                  ; not virtual terminal
```

Chapter 14
Software Interrupt System

AMOS provides a software interrupt service through which software can be written to respond to events in an asynchronous fashion. By allowing you to code your program so it can deal with events as they arise, the software interrupt system can greatly simplify your design and implementation.

Some of the features offered by the software interrupt system are:

- You can receive software interrupts caused by keyboard input, receipt of ITC messages, and timer events.
- Each type of software interrupt is separately maskable.
- One program can signal another by generating software interrupts.
- You can put your job into a wait state pending a software interrupt.

OVERVIEW OF THE SOFTWARE INTERRUPT SYSTEM

There are two different levels of software interrupts supported by AMOS: system-level interrupts and application-level interrupts. Application-level interrupts consist of four different types: keyboard input interrupts, ITC message interrupts, timer interrupts, and user interrupts. Keyboard input interrupts allow you to respond to an individual keystroke without having to check to see if characters are available. Likewise, ITC message interrupts allow you to respond to incoming messages as they arrive regardless of what else is going on. Timer interrupts allow repetitive actions to be performed on a regular basis without constant checking of the system clock. User interrupts, of which four different levels are provided, allow signaling within a job or between jobs.

System-level interrupts are used by the operating system and should ***not*** be modified at the application level.

For a program to use the software interrupt system, it must first set up a software interrupt vector table containing the address of a routine to handle each software interrupt that will be used. This routine will automatically be called each time a software interrupt occurs. Once set up, this vector table, along with a mask specifying which interrupts will be accepted, is passed to AMOS via an SIMSK monitor call. After the SIMSK is issued, your program will continue to execute normally, except that any incoming interrupts will be serviced automatically by calling the appropriate interrupt handler routine.

Whenever a software interrupt occurs, (which can be signaled currently at the end of most monitor calls) control will be transferred to the software interrupt handler you specified for the type of software interrupt that has occurred. This handler must perform whatever operation is necessary to service the interrupt and, when done, return control to the main program via an SIRTN call. All registers are automatically saved before the interrupt handler is called, and are automatically restored by the SIRTN call. The register contents upon entry into a software interrupt routine are unpredictable.

SOFTWARE INTERRUPT MONITOR CALLS

AMOS provides eight monitor calls to support the software interrupt system.

SIMSK - Set Software Interrupt Enable Mask

The SIMSK call allows you to specify which software interrupts you wish to respond to. By allowing you to selectively mask which interrupts are active, you need only supply interrupt handlers for those application-level interrupts you wish to use.

The calling format is:

```
SIMSK    tbl, mask
```

tbl	Points to a table containing the addresses of the software interrupt handler routines. An address must be specified for all types of interrupts that will be enabled. This table contains 32 longword addresses.
mask	Specifies a longword treated as a bit mask which defines which software interrupt types are to be enabled. Setting a bit to one enables the interrupt type; for application-level software interrupts, setting a bit to a zero disables the interrupt type.

The interrupt handler address table is 128 bytes long with the following offsets defined for each interrupt type:

Symbol	Octal Value	Hex Value	Meaning
SI.TIN	0	0	Terminal input interrupt
SI.ITC	4	4	Incoming ITC message interrupt
SI.TIM	10	8	Software interrupt timer interrupt
SI.US1	14	C	User interrupt level 1
SI.US2	20	10	User interrupt level 2
SI.US3	24	14	User interrupt level 3
SI.US4	30	18	User interrupt level 4
SI.IPC	34	1C	IPC software interrupt
SI.ABT	40	20	Forced abort (reserved, not used)
SI.EXI	44	24	Requested abort
SI.TCP	50	28	Asynchronous AlphaTCP event interrupt
SI.TOP	100	80	System-level interrupt for terminal output buffering

The software interrupt enable mask is a 32-bit field with the following bits defined:

Symbol	Octal Value	Hex Value	Meaning
SI$TIN	20000000000	80000000	Terminal input interrupt
SI$ITC	10000000000	40000000	Incoming ITC message interrupt
SI$TIM	4000000000	20000000	Software interrupt timer interrupt
SI$US1	2000000000	10000000	User interrupt level 1
SI$US2	1000000000	8000000	User interrupt level 2
SI$US3	400000000	4000000	User interrupt level 3
SI$US4	200000000	2000000	User interrupt level 4
SI$IPC	100000000	1000000	IPC software interrupt
SI$ABT	40000000	800000	Forced abort (reserved, not used)
SI$EXI	20000000	400000	Requested abort
SI$TCP	10000000	200000	Asynchronous AlphaTCP event interrupt
SI$TOP	100000	8000	System-level interrupt for terminal output buffering

You can change the types of application-level interrupts that are enabled at any time by issuing another SIMSK call; there is no need to change the interrupt handler routine table unless one of the routines moves or you wish to select a different handler.

Setting the interrupt mask to zero disables all application-level software interrupts.

SIRTN - Return from Software Interrupt

The SIRTN call is used to return control to the main program after completion of a software interrupt service routine. The calling format is:

```
SIRTN
```

SIWAIT - Wait for Software Interrupt

The SIWAIT call allows you to suspend the running of your job until a software interrupt occurs. The calling format is:

```
SIWAIT
```

After issuing an SIWAIT call, your job will be placed in a J.SIW wait state until a software interrupt occurs. Note that if no software interrupts are enabled when you issue an SIWAIT call, your job will be hung.

SITIMR - Enable Software Interrupt Timer

The SITIMR call allows you to request a software interrupt after a specific period of time has elapsed. This is particularly useful when a certain operation must be performed on a periodic basis.

The calling format is:

```
SITIMR   value
```

Value is a 32-bit value that specifies the number of 100 microsecond ticks that should occur before a software interrupt on the timer channel is requested.

The length of each timer tick is the same as with the TIMER and SLEEP monitor calls.

Because of job scheduling and other system latencies, the actual period of time between issuing an SITIMR call and the software interrupt will always be some amount of time longer than that specified. This amount is not predictable and therefore should not be relied upon for critical real time operations.

SISET - Set Software Interrupt Request On

The SISET call allows you to request a software interrupt of a specific type. This can be used with the user defined software interrupts to perform signaling. When used across jobs, this call provides an inter-job signaling method.

The calling format is:

```
SISET    mask{, jcb}
```

mask	A 32-bit longword treated as a bitmask to specify the software interrupts that are to be requested. The bitmask is ORed with the existing value. The possible bits are defined in the section on SIMSK, above.
jcb	This optional argument indexes the JCB of the job for which the interrupt request is to be set. If no argument is specified, the interrupt is requested for the current job.

In addition to the interrupt types defined in the section on SIMSK, there is an additional type of interrupt that can be generated via SISET. By selecting the SI$ABT interrupt type, a non-interceptable abort can be generated. This abort does not go through the normal software interrupt or Control-C handlers, but instead immediately aborts the job to AMOS command level. When used with the optional JCB argument of SISET, this can be used to free "stuck" jobs. If issued without a JCB argument, it causes immediate "suicide" by aborting the current job.

SICLR - Clear Software Interrupt

The SICLR call allows you to clear a pending software interrupt. You can also use it to clear a pending interrupt on another job.

The calling format is:

```
SICLR    mask{, jcb}
```

mask	A 32-bit longword treated as a bitmask to specify the software interrupts that are to be cleared. The possible bits are defined in section 14.2.1.
jcb	This optional argument indexes the JCB of the job for which the interrupt request is to be cleared. If no argument is specified, the interrupt is cleared for the current job.

SISTS - Get Software Interrupt Enable Status

The SISTS call allows you to determine which software interrupts are currently enabled for your job. This is useful when you wish to issue an SIMSK call which preserves all currently enabled application-level software interrupts, or when it is not desirable to modify the software interrupt structure for already enabled types. The enable mask for the current job is returned in register D6.

The calling format is:

```
        SISTS
```

SIDIS - Disable Software Interrupts

The SIDIS call allows you to disable specific software interrupts.

The calling format is:

```
        SIDIS   mask
```

Mask specifies a 32-bit longword treated as a bit mask which defines which software interrupt types are to be disabled. Setting a bit to one disables the interrupt type.

SAMPLE PROGRAM

The program below gives an example of how to use software interrupts. This rather simple-minded program outputs a constantly scrolling pattern of characters out to the terminal. Once per second, the terminal bell is sounded. During this terminal output, anything entered on the terminal is buffered until a carriage return is entered, at which time all buffered output is displayed on the terminal. This program uses both the timer interrupt (for the bell) and keyboard input interrupts (for the keyboard input buffering).

```
;SITST          Sample Software Interrupt Program
;
        SEARCH  SYS
        SEARCH  SYSSYM
        SEARCH  TRM

        VMAJOR = 1.
        VMINOR = 0.
        VEDIT  = 100.

;Main Entry Point
SITST:  PHDR     -1,0,PH$REU   ; program header
        TRMRST   D1            ; get current terminal status
        ORW      #T$DAT!T$ECS,D1  ; set data mode w/o echo
        TRMWST   D1            ; set terminal status
        LEA      A3,LIN        ; index line buffer
        LEA      A6,IDX        ; index place to store pointer
        MOV      A3,@A6        ;   and store it
        LEA      A6,INT        ; index interrupt handler
        LEA      A1,INTTBL     ; index vector table
        MOV      A6,SI.TIN(A1) ; place routine address into table
        LEA      A6,TIMINT     ; index timer interrupt handler
        MOV      A6,SI.TIM(A1)
        SIMSK    INTTBL,#SI$TIN!SI$TIM
                               ; enable term input and timer
        SITIMR   #10000.       ; select interrupt once per second

;Start main loop which just outputs scrolling characters
10$:    MOV      #^O40,D1
20$:    TTY                    ; display some output
        INC      D1            ; get next character
        CMPB     D1,#^O176     ; all out of characters?
        BLOS     20$           ;   no - keep looping
        BR       10$           ;   yes - go reset it

;Handle incoming keyboard input interrupt
INT:    TIN      ; get the character
        CMPB     D1,#3         ; is it a Control-C?
        JEQ      ABORT         ;   yes - abort program
        LEA      A6,IDX        ; index current point in line buffer
        MOV       @A6,A3
        MOVB     D1,(A3)+      ; store new character in buffer
        MOV      A3,@A6        ; save pointer for next time
        CMPB     D1,#^O15      ; was that a carriage return?
        BNE      10$           ;   no - all done
        CLRB      @A3          ; terminate line
        LEA      A3,LIN        ; index the line
        CRLF
        CRLF
        TTYL    @A3            ; display the entered data
        CRLF
        SLEEP    #5000         ; wait for user to read it
        LEA      A6,IDX        ; reset line buffer pointer
        MOV      A3,@A6
10$:    SIRTN                  ; return to main program

;Handle timer interrupt
TIMINT: MOVB     #7,D1;        ; ring terminal bell
        TTY
        SITIMR   #10000.       ; get another interrupt in 1 second
        SIRTN                  ; return from interrupt

;Come here on Control-C
ABORT:  CRLF                   ; be neat
        EXIT

;Define interrupt vector table
INTTBL: BLKB     SI.SIZ        ; interrupt vector table

;Define buffer and pointer for keyboard data
LIN:    BLKB     200.
IDX:    BLKL     1

        END
```

Chapter 15 The Inter-Task Communication System

WHAT IS INTER-TASK COMMUNICATION?

The Inter-Task Communication (ITC) system is designed to provide a standardized and centralized program to handle all the communication that goes on between tasks. With the advent of networking software that allows communication between different computers, it is necessary that a standard method of handling that communication be available—hence the ITC system.

The ITC system provides a simple datagram service both between processes on the same computer and, via the use of the AlphaNET network software, between different computers. The only visible difference between sending a message to a process on the same computer and sending a message to another computer is in the destination address, and possibly in the speed of transmission.

Message delivery is performed on a "best effort" basis; that is, there is no guaranteed delivery of messages. For this reason, all delivery verification and handshake protocols are left to upper level software. This method was chosen because of the generally high level of communications reliability within a single system or on a local network, and because of the many different application that will be using the message system, each requiring different levels of delivery verification and recovery.

GENERAL CONCEPTS

Before a task can send or receive a message, it must first open a message socket. This socket tells the ITC system which task is requesting access, and sets up a queue where pending messages can be stored. When a task requests a socket, it must give information, such as the size of the largest message to be used, and the maximum number of messages that can be queued for receipt at any one time. The message socket can be thought of as nothing more than a simple FIFO (first-in, first-out) queue.

A message is placed into a task's socket by AMOS whenever a message is received from some other task. Each incoming message corresponds to one entry in the socket, with entries being removed as they are serviced. If there is insufficient room when a message arrives, the request to send the message is denied.

When a task requests to read a message, it receives either the first pending message in the queue, or the next incoming message (if the queue is empty).

A task can process messages either synchronously or asynchronously. Synchronous processing requires that a task request each message in its proper turn. In this mode, the task will not know that a message is available to read unless it explicitly checks for a message.

Asynchronous processing involves a software "interrupt" which notifies the task of each incoming message. In order to process messages in this way, a task must have a Message Service Routine (MSR). An MSR is part of a task's program code, with its entry point being given to the software interrupt system via a SIMSK monitor call.

A task's MSR is given control whenever the MSR is enabled and the socket queue is not empty (at least one message has arrived and not yet been read), and the task is in a runnable state.

When the software interrupt occurs, the current program counter and status are stored, and the MSR is called. When the MSR is completed, control is returned to the point of task interruption via the SIRTN instruction.

Both the socket and the MSR can be turned on or off by the task itself. In addition, an MSR is turned off when the MSR is entered, preventing another message interrupt from occurring during the processing of the first message.

When a socket is disabled, requests to send a message to that socket are denied, but messages already in the socket queue remain unaffected. When an MSR is disabled, the MSR will not be called, even when messages are pending in the queue, although synchronous receipt of messages can still be performed.

More details on the software interrupt system can be found in Chapter 14.

TRANSPORT REQUIREMENTS

The ITC system does not care about the details of the underlying transport hardware and protocols. It is compatible with point-to-point, broadcast, token-passing, and other means of transport. There must, however, be some method for broadcasting messages, either by using the broadcast nature of the hardware (such as with a bus-oriented system), or by forwarding the message to the next node.

MESSAGE FORMAT

All messages have the same general format—a fixed size header followed by a variable length data area. The data area currently may range from 0 to 64K bytes, even though most messages will be less than 1K in size. In some cases, the configuration of a network or networks may restrict the data area to less than 64K.

The content of the data area is entirely transparent. This means that you may format it any way you want within your application programs.

Messages are formatted the same way whether they are being sent or received. This makes it easy to forward messages.

The format of the message header is:

0	Flags	MS.FLG
2		
4	Source address	MS.SRC
6		
10		
12	Destination address	MS.DST
14		
16	Message length (including header)	MS.SIZ
20	PPN of sender	MS.PPN
22	Privileges of sender	MS.PRV
24	Message code	MS.COD
26		MS.DAT

Message Flags

A 16-bit flags field is present in every message to define the characteristics of the message and to provide information as to how to best handle the message. The following flags are defined:

Flag	Meaning
MS$PRI	This is a high-priority message. When a message with this bit set is received, the message will be placed at the front of the pending message queue. This bit is intended to be used to flag messages affecting network control. It is not recommended that this bit be used for any other purpose.
MS$NOJ	This message does not require a job context to be sent. Normally, a job must have a socket opened prior to sending a message. In addition, AMOS automatically fills in the source address field within the message, guaranteeing that there is a return address available to the destination node. There are cases, however, where one or both of the actions is not desirable. Examples are interrupt service routines which need to send a message, but do not have a job context which would allow a socket to be opened; or a message forwarding service that must pass a message on to another destination, but does not wish the original return address modified.
MS$NTF	This message is being transmitted by NETFAM. This flag is for internal use only.

Source and Destination Addresses

Network addresses specify a specific process or group of processes within a group of networks. To do so, the message must specify the network, group of CPU's within the network, CPU within the group, and process within the CPU. Network addresses are a 48-bit field, formatted:

Decimal Bit Numbers	Section	Notes (Values in Decimal)
40 – 47	NETWORK	This 8-bit field specifies the network on which the address is located. The values are: 0 Source network number assumed 1-254 The specified network 255 Broadcast to all networks
32 – 39	GROUP	This 8-bit field specifies the group within the network that is being addressed. The values are: 0 Source group number assumed 1-254 Specified group 255 Broadcast to all groups
16 – 31	NODE	This 16-bit field specifies the node within a given group that is being addressed. The values are: 0 Source node number assumed 1 - 65519 Specified node 65520 - 65533 Reserved node numbers 65534 Special network server. The process address field specifies which special server.
0 – 15	SOCKET	This 16-bit field specifies the socket within a given node that is being addressed. The values are: 0-65519 Specified socket 65520-65531 Reserved node numbers 65532 Directed to electronic mail (MALSER) 65533 Directed to printer spooler (LPTSPL) 65534 Directed to NETSER 65535 Broadcast to all sockets If the Node address field contains 65534, then the socket address field is interpreted as: 1 Network Control Server 2-65535 Reserved

The destination address is broken down into these groups (network, group, node and socket) to provide information about the destination and to make it easier to send messages to groups of tasks. By using the individual address fields we can select a given node and broadcast a message to all the tasks on that node. Or we could broadcast a message to all the nodes within a group by simply specifying the desired network and node, and using "wildcard" specifications in the remaining address fields.

Broadcasting is not supported under AMOS 2.0 and earlier versions.

Message Length

The 16-bit message length field contains the total length of the message in bytes, including the header.

PPN of Sender

The 16-bit word contains the Project-Programmer Number that the sender of the message is logged into, taken from the JOBUSR word in the sender’s JCB. This field is for the use of protection mechanisms.

Privileges of Sender

The 16-bit word contains the privileges of the sender of the message, taken from the JOBPRV word in the sender’s JCB. For use in protection mechanisms.

Message Codes

The message codes indicate what type of message this is. This is application dependent, as the ITC system ignores this field. Intended for communication applications, such as Electronic Mail.

Data Area

This area is entirely user/program definable.

MESSAGE SYSTEM MONITOR CALLS

OPNMSG - Open a Message Socket

Before a task can send or receive messages it must open a message system "socket" to tell AMOS that it is going to use the message system. A job may execute multiple OPNMSG calls if it needs to. Such an operation does not actually allocate another socket, but instead simply increments a "nesting count" allowing multiple OPNMSG and CLSMSG calls to nest properly. The format is:

```
        OPNMSG   addr, status
```

addr is the address of an argument block formatted as described below and ***status*** is a register in which 32-bits of status are returned. See the section “Status Return Codes,” below, for a list of status codes.

Message sockets are automatically closed when an EXIT call is executed. The format of the argument block is:

0	flags	OM.FLG
2	Maximum message length	OM.LEN
4	Maximum number of pending messages	OM.MAX
6 10	Reserved	OM.MSR

OPNMSG Flags

A 16-bit field treated as a bitmask, with the bits defined as follows:

Symbol	Meaning
OM$SKE	Enables the socket. If this bit is zero, the socket will not be enabled until a SETMSS call is executed.
OM$NTF	Enable NETFAM mode (internal use only).

Maximum Message Length

The 16-bit maximum message length field specifies the longest message which the task is prepared to receive. If a message is longer than the maximum length, it is ignored, and an error status is returned to the sending process.

Maximum Number of Pending Messages Allowed

This 16-bit field specifies the maximum number of messages that may be queued for this task at any given time. By setting an upper limit, you can avoid buffer overflow problems.

CLSMSG - Close a Message Socket

Once a process is done sending or receiving messages, it should close its message system "socket" to notify the system that it does not wish to receive any further messages. As multiple OPNMSG and CLSMSG calls can be executed by a single job, a "nesting count" is needed. This count is incremented each time an OPNMSG is performed, and decremented each time a CLSMSG call is performed. Only when this "nesting count" reaches zero is the socket truly closed. The format is:

```
CLSMSG   status
```

status is a register in which 32 bits of status are returned. Values are defined in the section “Status Return Codes,” below.

Message sockets are automatically closed when an exit call is executed.

SNDMSG - Send a Message

The format for the SNDMSG call is:

```
SNDMSG   buffer address, status, flags
```

buffer address	The address of the buffer that is to send the outgoing message header and data.
status	A register in which 32 bits of status are returned (see the section “Status Return Codes,” below).
flags	Flags that affect the sending of the message. Reserved for future expansion.

The message header and data area should be set up prior to using the SNDMSG call. You must supply data in the flag (MS.FLG), destination address (MS.DST), and message size (MS.SIZ) fields. The SNDMSG call will automatically fill in (and overwrite) the source address (MS.SRC), source PPN (MS.PPN), and source privileges (MS.PRV) fields.

The SNDMSG call may be performed from within an interrupt service routine.

RCVMSG - Receive a Message

Used to receive a message or to check for pending messages. The format is:

```
          RCVMSG   buffer address, status, flags
```

buffer address	The address of the buffer that is to receive the incoming message header and data.
status	A register in which 32 bits of status are returned (see the section “Status Return Codes,” below).
flags	Flags that affect the receipt of the message:

Symbol	Meaning
MS$WAT	If no messages are available, place the job in a wait state. If not set, the call will return immediately.

You must supply a buffer area large enough to accept the largest expected message. No checking for buffer overflow is performed. RCVMSG will return the message with the flag (MS.FLG), source address (MS.SRC), source PPN (MS.PPN), source privileges (MS.PRV) and message size (MS.SIZ) fields filled in.

CHKMSG - Check for Received Messages

CHKMSG is used to determine the number of messages the current task has pending. The format is:

```
          CHKMSG   arg
```

Arg is a field to contain the number.

In addition to the number returned, the condition code Z-bit will be set if no messages are pending, and reset if there are messages pending. This will occur even if arg is not specified in the call.

WTMSG - Wait for Receipt of Message

WTMSG allows a task to wait for a message to arrive. provisions are made for allowing escape from the wait condition if no message is received within a specified period of time. When no MSR is in use, this call makes it convenient to wait for the next message.

The format is:

```
WTMSG   arg
```

Arg is the maximum number of 100 microsecond intervals to wait. The call will return when either a message has been received, or arg intervals have passed. If arg is zero, the call will wait until a message is received.

Upon the return from the WTMSG call, the Z-bit will be set if the return was caused by the receipt of a message, and will be reset if the return was caused by a time-out condition.

SETMSS - Set MSQ/MSR Status

Allows a task to set/reset the status of both the socket and the message service routine. Each may be enabled or disabled. If the socket is disabled, no messages may be received. If the Message Service Routine (MSR) is disabled, no software interrupts will be generated by the ITC system, and messages must be received synchronously. The format is:

```
SETMSS  flags
```

Flags specifies which facilities are to be turned on after the call is executed. Set the flag to 1 to enable or 0 to disable. The flags are:

Symbol	Meaning
MS$SKE	Enable the socket to receive
MS$MSE	Reserved for future use

Status Return Codes

The various message system monitor calls return status codes:

Symbol	Octal Value	Hex Value	Meaning
M$ENSK	1	1	No socket is open (by OPNMSG) for the sending process
M$ENMP	2	2	No messages are pending
M$ENMS	3	3	Operating system does not support the message system
M$EDNN	4	4	Destination network does not exist
M$ENNN	5	5	Destination node does not exist
M$ESNN	6	6	Destination socket does not exist
M$ESAE	7	7	Socket already exists
M$ENMB	10	8	No message buffers are available
M$ENQB	11	9	No queue blocks are available
M$EAOP	12	A	Argument address is outside of caller's partition
M$EDSF	13	B	Destination socket is full
M$EDSN	14	C	Destination socket is not enabled
M$EMTL	15	D	Message length is greater than the destination's maximum
M$EATO	16	E	ACK time-out

NETWORK DRIVERS

Whenever the ITC system gets a request to send a message to a task not located on the current system (the network, group, and/or node fields in the destination address are non-zero), the request is forwarded to a Network Driver (.NDV), providing one is found that matches the network address. It is the NDV's responsibility to forward the message to the destination over whatever communications link it supports.

Whenever the ITC system receives a request for a remote system, it scans through an internal table (the Network Definition Table) to determine which Network Driver should receive the request. This internal table is defined at system initialization by the NETINI program. Each entry in the table describes a different type of network connection.

Optionally, the ITC system may also scan through an internal list of active nodes to determine if the destination is currently available. Not all networks can support the node list—it is controlled by the NODECHECK parameter in the network initialization file (NET.INI).

THE NETWORK DEFINITION TABLE

The Network Definition Table contains one entry for each network available on the system. It is set up by the NETINI program during system initialization. The format of each definition table is:

Offset	Field	Symbol
0, 2	Pointer to next table entry	NT.NXT
4, 6	Network address field	NT.NET
10	Flags	NT.FLG
12, 14	Network driver name	NT.NAM
16, 20	Pointer to network driver	NT.DVR
22, 24	Pointer to node list	NT.NLS
26, 30	Gateway forwarding address	NT.GTW
32 … 60	Symbolic name of network	NT.SYM

Pointer to Next Table Entry

The Network Definition Table is maintained as a linked list, using this longword entry to link to the next entry in the table. The list is terminated by a zero entry in this field.

The Network Address Field

The network address field longword contains the network address corresponding to the current node on this network. The "network" field within this address specifies the network number for this network. The "group" and "node" fields of this address specify the address by which the current node is known on this network.

This field is required because a single node will have multiple addresses when it resides on more than one network.

The Flags Longword

The flags longword contains flags which describe the characteristics of this network. The flags are:

Symbol	Meaning
NT$NDC	This network does not need node validation by scanning through the node list. The setting is determined by the NODECHECK parameter in the NETINI initialization file.
NT$GTW	This network is a gateway forwarding connection. All messages for this network should be sent to the gateway address contained in NT.GTW.

The Network Driver Name

The network driver name longword contains the name of the network driver associated with this network, packed in RAD50 format.

The Pointer to the Network Driver

The pointer to the network driver is a longword containing a pointer to the network driver associated with the current network. The format of the network driver is described in “The Network Driver Structure,” below.

The Pointer to the Node List

The pointer to the node list longword contains a pointer to the Node list associated with the current network. If no nodes are currently active, the pointer will contain a zero.

Gateway Forwarding Address

For gateway connections, the gateway forwarding address field contains the network address of the node responsible for forwarding gateway traffic. This field is only present when AlphaNET 2.0 or later is in use.

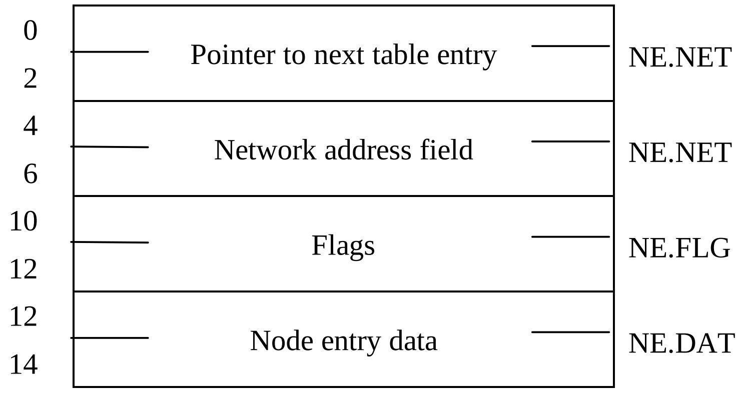
Network Symbolic Name

The 22-byte network symbolic name field contains a zero-terminated ASCII string giving a symbolic name for this network. This name is used only for descriptive purposes, such as when displaying network status. This field is only present when AlphaNET 2.0 or later is in use.

THE NODE LIST STRUCTURE

The Node list data structure is used to maintain a dynamic list of the nodes currently available on a network. Not all networks can support this function, however, so this data structure must be treated as an optional item. No software should ever depend on its presence, or the software will not be network independent.

The Node list for a given network can be located by using the Node list pointer entry (NT.NLS) in the Network definition table. Each entry in the Node list is formatted as:



The Pointer to the Next Entry

The Node list is maintained as a linked list using this longword to point to the next entry in the list. A zero in this field terminates the list.

The Network Address Field

The network address longword contains the network address, including network, group, and node numbers that uniquely identify the current node.

The Flags Word

The flags word contains flags which describe the characteristics of this driver. The flag is defined as:

Symbol	Meaning
NL$ALC	This entry is allocated. This flag must be checked prior to relying on any of the information in this in this table entry, as there may be table entries linked into the data structure which are not currently active. The Node list entry is only valid if this flag is set.

The Node Entry Data Area

The node entry data area longword can be used to store network dependent information on a node by node basis. If additional space beyond one longword is needed, additional space can be allocated by requesting it by using the Node Entry Data Size (ND.NES) parameter of the Network Driver, described below. Any extra space allocated in this fashion is in addition to the single longword present in all Node list entries.

THE NETWORK DRIVER STRUCTURE

Each NDV is a simple collection of subroutines packaged into a single file. In this sense it is very similar to a terminal driver (TDV), or an interface driver (IDV). Like these other drivers, it resides in DSK0:[1,6]. It is loaded into system memory by the NETINI program, and thus must be re-entrant (sharable).

Each of the routines contained in the NDV performs a separate function. In addition to these routines, each NDV contains descriptive information which other processes may examine to determine some of the characteristics of the NDV itself. The header block of each NDV contains:

0 2	Flags	ND.FLG
4 6	Initialization entry point	ND.INI
10 12	Send message entry point	ND.WRT
14 16	Special function entry point	ND.STS
20 22	Hardware address	ND.HWA
24 26	Interrupt vector address	ND.INT
30	Node entry data area size	ND.NES

The Flags Longword

This longword contains flags which describe the characteristics of this driver. The flags are defined as:

Symbol	Meaning
ND$VID	This is a VideoNET driver
ND$SLW	This is a "slow" driver. This bit is used by the ITC system to adjust some of its time-out parameters for networks that are slower than normal. For example, you might have an asynchronous dial-up network, where the action of dialing the telephone and waiting for a response could take much longer than the usual time-out constants would allow. When this flag is set, the normal 20 second ITC time-out is extended to 60 seconds.

The Initialization Entry

The initialization entry routine is called by the NETINI program immediately after the network has been defined. It can perform such functions as initializing hardware, notifying other nodes that the current node is now available, and allocating buffer space. The following registers are set up:

Register	Purpose
D0	Node count. This is the value specified in the NODECOUNT option within the network initialization file being processed by NETINI.
D2	Contains device address passed by NETINI. This number is the number immediately following the DEVICE= statement in the network initialization file or is zero if no DEVICE= statement is in the file.
A3	Points to the network definition block for the network currently being initialized.
A4	Points to the next available word in system memory. To allocate additional buffer space, start at the word pointed to by this register. After the space has been allocated, update the MEMBAS pointer.

When the initialization routine returns to the caller (using RTN), the network driver must return the updated system memory base pointer in A4.

The initialization routine must preserve all registers other than D6, D7, and A6.

The initialization routine will be executed with the booting job's context, in User Mode.

The Send Message Entry

This routine is called whenever a request to send a message to a node on the current network is received. It is the responsibility of this routine to deliver the message to the specified remote node.

The Send Message routine may do anything, from directly transferring the message to a hardware interface, to simply forwarding the message to another task for processing. The following registers are set up:

Register	Purpose
D1	Contains the flags from the message to be sent
A3	Points to the network definition block for the network currently being initialized
A4	Points to the message to be sent

When the Send Message routine returns to the caller (using RTN), the following return code is handled:

Register	Purpose
D1	The return code to be returned to the sending process

The routine must preserve the contents of A3, D2, D3, D4, and D5. All other registers may be altered.

The routine will be executed with the sending job's context, in Supervisor Mode.

The Special Function Routine Entry

Reserved for future expansion.

The Hardware Address

The hardware address entry contains the 32-bit physical device address of the hardware associated with the network. It provides information to other processes, but is not used by the ITC system itself.

The Interrupt Vector Address

The interrupt vector address entry contains the address of the routine that handles any interrupts associated with the current network hardware. Provides information to other processes, but is not used by the ITC system itself.

The Node Entry Data Size

The node entry data size entry contains the size of the extra data space to be allocated for each node entry in the network node list. The extra space can be used to store special network information on a per-node basis.

Chapter 16
Serial Communications System

AMOS provides a number of services and calls to support serial communications. While most of these can be accomplished through the normal terminal service routines, the serial communications system provides higher performance alternatives, specifically tailored to the task of serial communications, eliminating much of the overhead normally associated with the terminal handling that AMOS must perform. Moreover, it does this in a device independent fashion.

Some of the features offered by the serial communications system are:

- Transparent, hardware independent access to the modem control signals (CTS, DSR, RI, etc.) present on the majority of serial interfaces.
- Hardware independent support of modem functions such as dialing, answering, etc.
- Support for very high-speed serial I/O by completely bypassing all TRMSER support functions.

Traditionally, gaining access to the control signals required in data communication applications required software developers to access the hardware directly, defeating the device independent nature of AMOS, and creating potential incompatibilities when a variety of software and hardware is used.

In addition, some high-performance applications require very high-speed communications support. Again, implementing a solution to this type of problem often created incompatibilities and necessitated redundant software development.

In keeping with AMOS convention, symbols in this chapter containing "." specify memory offsets. Symbols containing "$" specify a bit mask for logical instructions (AND, OR, etc.). In addition, for each symbol containing "$" there is a complementary symbol containing "%" to specify bit positions for bit instructions (BTST, BSET, etc.).

COMSET - SET COMMUNICATIONS PORT PARAMETERS

The COMSET call allows you to set various communications parameters associated with a particular communications port.

The calling format is:

```
COMSET   port, addr
```

port	Points to the terminal definition block defining the communications port which is to be affected.
addr	Indexes a 12-byte argument block in memory specifying the parameters to be set.

The 12-byte argument block is formatted into the following five fields:

Symbol	Meaning
CM.BAU	This 16-bit field contains the desired baud rate, defined as follows:

Octal Value	Hex Value	Baud Rate	Octal Value	Hex Value	Baud Rate
0	0	50 baud	12	A	2000 baud
1	1	75 baud	13	B	2400 baud
2	2	110 baud	14	C	3600 baud
3	3	134.5 baud	15	D	4800 baud
4	4	150 baud	16	E	7200 baud
5	5	200 baud	17	F	9600 baud
6	6	300 baud	20	10	19200 baud
7	7	600 baud	21	11	38400 baud
10	8	1200 baud	22	12	57600 baud
11	9	1800 baud			

CM.DAT This 16-bit field specifies the number of data bits to be used by the communications port, encoded as follows:

Value	Data Bits
0	5 data bits
1	6 data bits
2	7 data bits
3	8 data bits

CM.STP This 16-bit field specifies the number of stop bits to be used with this communications port

Value	Stop Bits
0	1 stop bit
1	1.5 stop bits
2	2 stop bits

CM.FLA This 16-bit field specifies miscellaneous flags to be used in setting up the communications port:

Flag	Meaning
CM$PAR	If set, parity checking/generation is enabled. If zero, no parity checking or generating is performed.
CM$ODD	If set, and CM$PAR is also set, odd parity is used. If zero, and CM$PAR is also set, even parity is used. If CM$PAR is zero, this flag is ignored.
CM$TRTS	If set, and the feature is available on the selected interface, this flag enables transmitter RTS control. Normally, this feature toggles RTS each time the transmitter is disabled. Because AMOS frequently disables the transmitter, this feature is not always desirable. Setting this flag turns this feature on.
CM$RRTS	If set, and the feature is available on the selected interface, this flag enables receiver RTS control. When enabled, the RTS line will be toggled whenever the receiver FIFO buffer is full, preventing further transmission that might result in an overrun condition, providing the transmitting end has the CM$CTS feature enabled.
CM$CTS	If set, and the feature is available on the selected interface, this flag enables transmitter CTS control. When enabled, and the CTS line is de-asserted, the transmitter will be disabled until CTS returns to an active state.
CM$BRK	If set, and CM$BKE is also set, a break condition will be placed on the transmit data line. The break condition will continue until this bit is reset.
CM$BKE	This bit, when set, enables the operation of the CM$BRK flag. If CM$BKE is not set, the setting of CM$BRK is ignored.

Symbol	Meaning
CM.STS	This 32-bit field defines an "interrupt mask" to be applied against the modem status lines. Each of the possible inputs is assigned a bit in this longword. If a bit is a one in this field, then the status change interrupt routine will be called anytime the control line associated with the input bit changes state. The program should use the COMRST call prior to setting up this longword to make sure that the desired input status line is available on the particular communications port in question. Note that most of the symbols below will ***not*** generate interrupts on Alpha Micro interfaces. Symbols have been defined as follows:

Symbol	Octal Value	Hex Value	Meaning
RS$PGD	2	2	Protective ground
RS$TXD	4	4	Transmit data
RS$RXD	10	8	Receive data
RS$RTS	20	10	Request to send
RS$CTS	40	20	Clear to send
RS$DSR	100	40	Data set ready
RS$SGD	200	80	Signal ground
RS$DCD	400	100	Data carrier detect
RS$009	1000	200	RS-232 pin 9
RS$010	2000	400	RS-232 pin 10
RS$011	4000	800	RS-232 pin 11
RS$SRL	10000	1000	Secondary receive line signal
RS$SCT	20000	2000	Secondary clear to send
RS$STX	40000	4000	Secondary transmit data
RS$TSC	100000	8000	Transmitter signal element timing (DCE)
RS$SRX	200000	10000	Secondary receive data
RS$RSC	400000	20000	Receiver signal element timing (DCE)
RS$018	1000000	40000	RS-232 pin 18
RS$SRT	2000000	80000	Secondary request to send
RS$DTR	4000000	100000	Data terminal ready
RS$SQD	10000000	200000	Signal quality detector
RS$RGI	20000000	400000	Ring indicator
RS$DSS	40000000	800000	Data signal rate selector (DCE/DTE)
RS$TST	100000000	1000000	Transmitter signal element timing (DTE)
RS$025	200000000	2000000	RS-232 pin 25

This call returns with the Z-bit set if it was successful; the Z-bit will be returned reset if any error condition was detected. If the error is of a catastrophic nature (e.g., invalid port number, port does not exist, etc.), the N bit will also be set.

The symbol CM.SZE defines the size of the argument block used by COMSET.

COMINT - SET INTERRUPT SERVICE VECTORS

One of the major benefits of the serial communications system is the ability to get individual interrupt service for input characters, output characters, and status change. Interrupt service for these three cases can be provided by the user when invoking the COMINT call by specifying an address for a routine to be associated with each of them.

By transferring control directly to the application, with a minimal amount of system overhead, special purpose applications can provide maximum throughput. This capability is provided by allowing the interface driver (IDV) to transfer control directly to a user supplied routine rather than performing normal terminal buffering via the terminal service system and the TRMICP and TRMOCP calls. The user supplied routine can then process the characters in a fashion tailored to the specific application, rather than having to "make do" with the standard terminal service routines which must try to be all things to all programs.

Before any of the three user routines can be called, the application program must inform the system of the address of the interrupt routines. This is done via the COMINT monitor call. A program may make use of as few as zero and as many as three of the three available interrupts. If an interrupt is not to be used, the program should supply a zero as the interrupt service address.

The price paid for this extra flexibility and performance is simply the burden it places on the applications programmer. The programmer must be concerned with the details of character buffering and timing considerations, items normally left up to the system software itself. For this reason, programming with the interrupt service routines is best left to experienced assembly language systems programmers.

Particular care must be taken with the amount of time spent in the user supplied interrupt service routines. Because these routines are executed with all other interrupts disabled, any time spent in the service routines defers all other interrupts, potentially having a major impact on system performance. In extreme cases, spending too long in the interrupt service routine can lead to complete system failure. Try to limit interrupt service time to less than 100 microseconds. It is not possible to enable other interrupts while in a communications service interrupt routine; trying to do so will lead to system lockups.

The calling format is:

```
            COMINT   port
```

port points to the terminal definition block defining the communications port for which interrupts are to be serviced.

At the time of the call, the following registers must be set up:

Register	Purpose
A1	Contains the address of the output character interrupt routine, or zero if normal TRMOCP processing by the IDV is desired.
A2	Contains the address of the input character interrupt routine, or zero if normal TRMICP processing by the IDV is desired.
A3	Contains the address of the status change interrupt routine, or zero if no status change interrupt is desired.

Because control is being given to the user program at interrupt level, it is very important that all interrupt service routines used with COMINT be carefully optimized and debugged to make sure that the system is not adversely affected. Maximum interrupt service time should be kept to 100 microseconds or less.

Also keep in mind that because the service routines are running at interrupt service level, without a job context, very few monitor calls are valid. Most all of the monitor calls require a job context (such as all I/O calls) making them illegal in interrupt service routines. Any attempt to use such calls will result in the system crashing.

The applications program must obey all rules concerning the use of the TCB semaphore when accessing the contents of the TCB. Failure to do so may result in system lockups due to the multi-processing nature of terminal handling under AMOS. Refer to Appendix B of this book for more information on handling the TCB.

It is up to the applications program to disable the interrupt service addresses before exiting. Refer to the *AM-350 Intelligent I/O Controller Installation Instructions*, PDI-00350-00, for more information related to the COMINT call.

The Input Character Routine

The input character interrupt routine is called each time a character is received by the communications port hardware. The input routine is called with the input character contained in the low-order byte of D1. No modification has been done to the character at this point.

In addition to the incoming character, the following registers are passed to the input character interrupt routine:

Register	Meaning	
D6	This register contains the receiver status that reflects the condition of the receiver after receiving the current character. The flags contained in this register are:	
	Symbol	**Meaning**
	CS$FRM	If set, a framing error was detected.
	CS$ORN	If set, an overrun error was detected.
	CS$PAR	If set, a parity error was detected.
	CS$BRK	If set, a break condition was detected. This does not indicate whether or not a break is still active, but indicates a break was received sometime after the last time receiver status was reported.
A6	This register contains the address of the terminal control block assigned to the port. Your routine should not modify the contents of A5.	

The only registers that have been saved are D1, D6, D7, and A5. It is the responsibility of the input character interrupt routine to save any other registers it may need to use.

The Output Character Routine

The output character interrupt routine is called each time the communications port hardware is ready to accept another character for output.

The following register is passed to the routine:

Register	Purpose
A5	This register contains the address of the terminal control block assigned to the port. Your routine should not modify the contents of A5.

The output character routine should supply the next character available for output in the low order byte of D1. If no further characters are available, the output character routine must place a longword -1 in D1.

You must use the TINIT call to start up output processing. Normally, the output character routine will only be called after the output process has been initiated by a TINIT call. However, your routine should be prepared to return -1 in D1 any time it is called and there are no characters, even if the output process has not been started up.

The only registers that have been saved are D1, D6, D7, and A5. It is the responsibility of the output character interrupt routine to save any other registers it may need to use.

The Status Change Routine

The status change interrupt routine is called whenever one or more of the status bits specified in the status change mask (specified in the COMSET monitor call) changes state.

The following registers are passed to the routine:

Register	Purpose
A5	This register contains the address of the terminal control block assigned to the port. Your routine should not modify the A5 register.
D6	Contains the current status of the RS-232 input pins for the port. (A bit value of 1 indicates a true or asserted state.)
D7	Available pin mask, which specifies the RS-232 input and output pins available for the port.

The only registers that have been saved are D1, D6, D7, and A5. It is the responsibility of the output character interrupt routine to save any other registers it may need to use.

Initializing Terminal Output (TINIT)

The TINIT call is used to initiate output through a communications port. Whenever a character is placed in the output character buffer (to be picked up by the output character interrupt routine), but no output is currently taking place (the T$OIP flag is not set in the terminal status word), then a TINIT call must be performed to initiate character output. To make programming easier, the TINIT call can be issued for every character entered into the user's output buffer, although this is obviously less efficient than only doing it when T$OIP is not set.

After a character has been placed in the output buffer, and a TINIT call issued, an output character interrupt will take place, allowing the interrupt service routine to remove the character from the buffer and deliver it to the hardware for output. Any time that an output character interrupt occurs but there are no characters pending output, the T$OIP flag will be cleared and the output character interrupt will be dismissed, not to recur until a later TINIT starts the process over again.

The calling format is:

```
                INIT     port
```

port is a 32-bit address pointing to the terminal definition block defining the communications port which is to be affected.

COMRST - READ COMMUNICATIONS STATUS LINES

The COMRST call allows a program to read the status of all possible incoming modem control signals, as well as to obtain a bit mask specifying which of these incoming modem control signals are available on the particular hardware in use.

The calling format is:

```
COMRST  port, stat, mask
```

port	A 32-bit pointer to the terminal definition block defining the communications port which is to be read.
stat	A 32-bit register in which the status of all input status bits will be returned. Any bit set to a one indicates that the corresponding signal is asserted at the interface.
mask	A 32-bit register in which the valid status signals will be flagged. If a given status signal is available from the hardware in use, the corresponding bit will be a one. Thus if the current hardware has valid DSR status, the bit RS$DSR of this mask will be a one. If valid status is not available, the bit will be a zero in this mask. Symbols for the bits in this mask are defined in the section on COMSET, above.

COMWST - WRITE COMMUNICATIONS STATUS LINES

The COMWST call allows a program to control the status of all possible outgoing modem control signals. Before attempting to set a given bit, however, the program should make sure that the hardware in use of the particular communications port supports the status bit in question. A mask describing all supported bits can be obtained via the COMRST call.

The calling format is:

```
COMWST  port, stat
```

port	A 32-bit pointer to the terminal definition block defining the communications port which is to be affected.
stat	A 32-bit register in which you supply the status of the status bits you wish to output. Any bit you set to a one in this register will be sent to the hardware to be asserted. Symbolic definitions of the bits used in this parameter are as follows:

Symbol	Octal Value	Hex Value	Meaning
RS$RTS	20	10	Request to send
RS$DTR	4000000	100000	Data terminal ready

MDREQ - REQUEST A MODEM

Before a modem and communications port can be used by a program, it must first be assigned to that program. Likewise, in a system with multiple modems, a program may not be able to effectively use all of the modems, but may have special requirements. The MDREQ call arbitrates this reservation system for modems. By specifying the type of modem being requested (including "any") and by keeping track of the modems currently available, the MDREQ call assigns modems on a first-come first-served basis.

The calling format is:

```
        MDREQ    type, port
```

type — A 32-bit argument specifying the type of modem being requested. A zero in this field means any modem is acceptable. Other values for this field are defined as follows:

Symbol	Octal Value	Hex Value	Meaning
MD$B30	1	1	300 baud supported
MD$B12	2	2	1200 baud supported
MD$B24	4	4	2400 baud supported
MD$B48	10	8	4800 baud supported
MD$B96	20	10	9600 baud supported
MD$B19	40	20	19200 baud supported
MD$B38	100	40	38400 baud supported
MD$B56	200	80	57600 baud supported
MD$ASY	200000	10000	async modem
MD$DIL	400000	20000	autodialer available

By setting one or more of these bits, a program can specify that it is only willing to accept a modem having the specified capability. For example, a program which requires 9600 baud transmission capability can select that bit, ensuring that no slower modems will be assigned by the MDREQ call. By performing multiple MDREQ calls, reducing the requirements each time the call returns unsuccessfully, a program can obtain the fastest (or most capable) modem currently available.

port — A 32-bit address register in which the address of the terminal definition block of the communications port to which the assigned modem is returned.

This call returns with the Z-bit set if it was able to assign a modem, reset if not.

MDRTN - RETURN A MODEM

The MDRTN call is used in conjunction with MDREQ to arbitrate the assignment of modems.

The calling format is:

```
        MDRTN    port
```

Port is an address register containing the address of the terminal definition block of the communications port to which the modem being returned is connected.

MDSET - SET MODEM COMMUNICATIONS PARAMETERS

The MDSET call allows you to set various communications parameters associated with a particular modem. Note that this call changes the *modem* settings, not the *communications port* settings. The communications port parameters must be changed by using the COMSET call.

The calling format is:

```
MDSET   port, addr
```

port	A 32-bit pointer to the terminal definition block defining the communications port which is to be affected.
addr	A 32-bit pointer to an 8-byte argument block in memory specifying the parameters to be set.

The 8-byte argument block is formatted into the following four fields:

Symbol	Meaning
CM.BAU	This 16-bit field contains the desired baud rate, defined as follows:

Octal Value	Hex Value	Baud Rate	Octal Value	Hex Value	Baud Rate
0	0	50 baud	12	A	2000 baud
1	1	75 baud	13	B	2400 baud
2	2	110 baud	14	C	3600 baud
3	3	134.5 baud	15	D	4800 baud
4	4	150 baud	16	E	7200 baud
5	5	200 baud	17	F	9600 baud
6	6	300 baud	20	10	19200 baud
7	7	600 baud	21	11	38400 baud
10	8	1200 baud	22	12	57600 baud
11	9	1800 baud			

Symbol	Meaning
CM.DAT	This 16-bit field specifies the number of data bits to be used by the communications port, encoded as follows:

Value	Data Bits
0	5 data bits
1	6 data bits
2	7 data bits
3	8 data bits

Symbol	Meaning
CM.STP	This 16-bit field specifies the number of stop bits to be used with this communications port.

Value	Stop Bits
0	1 stop bit
1	1.5 stop bits
2	2 stop bits

Symbol	Meaning
CM.FLA	This 16-bit field specifies miscellaneous flags to be used in setting up the communications port:

Symbol	Meaning
CM$PAR	If set, parity checking/generation is enabled. If zero, no parity checking or generating is performed.
CM$ODD	If set, and CM$PAR is also set, odd parity is used. If zero, and CM$PAR is also set, even parity is used. If CM$PAR is zero, this flag is ignored.

MDSET returns with the Z-bit set if it was successful; the Z-bit will be returned reset if any error condition was detected.

MDDIAL - DIAL A MODEM

When using auto-dial modems it becomes necessary to use the dialer portion of the modem to place outgoing calls in a modem independent fashion. This can be done via the MDDIAL monitor call. The MDDIAL call allows you to both dial calls and to hang up.

The calling format is:

```
MDDIAL  port, addr, status
```

port	A 32-bit pointer to the terminal definition block defining the communications port to which the modem in question is connected.
addr	A 32-bit pointer to a null terminated (ASCIZ) string describing the number to be dialed, formatted as described below.
status	Specifies a 32-bit data register into which the status of the call will be placed. The status will be zero if the call was successful.

The Z-bit will be returned set if the call was successfully completed, reset on an error.

The phone number to be dialed must consist of a null terminated string consisting of the digits 0 through 9 and the following special characters:

Character		Purpose
,	(comma)	Pause in the dialing of the number, as when waiting for a second dial tone in PBX systems.
#	(pound)	Dial the special "#" symbol when tone (DTMF) dialing.
*	(asterisk)	Dial the special "*" symbol when tone (DTMF) dialing.
P		Commence rotary (pulse) dialing, if supported by the modem.
T		Commence tone (DTMF) dialing, if supported by the modem.
H		Hang up phone. Must be only character in command string.

The status codes returned by MDDIAL are as follows:

Symbol	Octal Value	Hex Value	Meaning
MR$OK	0	0	Call completed successfully
MR$UAF	1	1	Modem does not support requested feature (rotary or tone dialing)
MR$NDT	2	2	No dial tone was detected
MR$BSY	3	3	Number was busy
MR$NOA	4	4	No answer
MR$NOC	5	5	No carrier
MR$INV	6	6	Invalid/blacklisted number
MR$MNR	7	7	Modem not responding
MR$REL	15	D	Reliable connection

Bits 8 - 14 are the connected baud rates as defined in CM.BAU in the previous section.

Not all modems are capable of detecting all error conditions. In particular, many inexpensive modems have no provision for dial tone or busy signal detection and will therefore always return status code 4 when a call fails to be completed, regardless of the true reason.

MDON - ENABLING A MODEM

When used in dial-up communications environments it sometimes is desirable to be able to control whether a modem is available to be called into. This can be very important when designing the system security features to prevent unauthorized access. The MDON monitor call enables a modem to allow external systems to dial into it. A companion monitor call (MDOFF) disables this capability.

The calling format is:

```
        MDON    port
```

port is a 32-bit pointer to the terminal definition block defining the communications port to be affected.

If the call was successfully completed, the Z-bit will be returned set. If the call fails for any reason, the Z-bit will be reset to zero.

MDOFF - DISABLING A MODEM

A companion call to MDON, the MDOFF call allows a program to disable a specific modem from external access.

The calling format is:

```
        MDOFF   port
```

port is a 32-bit pointer to the terminal definition block defining the communications port which is to be affected.

If the call was successfully completed, the Z-bit will be returned set. If the call fails for any reason, the Z-bit will be reset to zero.

MODEM DRIVER FORMAT

Modem drivers are device specific modules which allow AMOS to control a modem while making the software interface to the modem hardware "modem independent." Modem drivers have the extension .MDV and reside in DSK0:[1,6].

Internally, modem drivers are structured much like other drivers used by AMOS. They contain a header area containing descriptive information and a dispatch table. It is not intended that this header area be directly accessed by programs which should instead use monitor calls to access the information.

The format of the modem driver is as follows, with the symbols defined below defined in SYSSYM.UNV:

Symbol	Size	Meaning
MD.HDR	10 bytes	Standard PHDR style file header defining version number and program header flags.
MD.TYP	4 bytes	Modem type number as used in the MDREQ call.
MD.SET	4 bytes	Entry point to the MDSET routine. This entry should contain a JMP to the routine which will be called each time an MDSET call is executed. Upon entry to the routine, A5 will index the terminal control block and A1 will index the argument block.
MD.DIL	4 bytes	Entry point to the MDDIAL routine. This entry should contain a JMP to the routine which will be called each time an MDDIAL call is executed. Upon entry to the routine, A5 will index the terminal control block and A1 will index the argument block.
MD.ON	4 bytes	Entry point to the MDON routine. This entry should contain a JMP to the routine which will be called each time an MDON call is executed. Upon entry to the routine, A5 will index the terminal control block.
MD.OFF	4 bytes	Entry point to the MDOFF routine. This entry should contain a JMP to the routine which will be called each time an MDOFF call is executed. Upon entry to the routine, A5 will index the terminal control block.
MD.SND	4 bytes	Entry point to the MDSND routine. This entry should contain a JMP to the routine which will be called each time an MDSND call is executed. Upon entry to the routine, A5 will index the terminal control block and A1 will index the argument block.
MD.RCV	4 bytes	Entry point to the MDRCV routine. This entry should contain a JMP to the routine which will be called each time an MDRCV call is executed. Upon entry to the routine, A5 will index the terminal control block and A1 will index the argument block.
MD.CON	4 bytes	Pointer to modem time constants. This longword offset (relative to the base of the modem driver) points to a table of timing constants used by this particular modem. These timing constants can be used to implement time-out error recovery techniques. Each constant is two bytes and specifies the maximum time to be taken by a particular call, in seconds, as follows:

Symbol	Call
MD.TST	MDSET
MD.TDI	MDDIAL
MD.TON	MDON
MD.TOF	MDOFF
MD.TSD	MDSND
MD.TRC	MDRCV

Symbol	Size	Meaning
MD.IMP	4 bytes	Specifies the number of bytes to allocate as impure space for this modem. Impure space is allocated at TRMDEF time for each port using a specific modem type. The impure space is pointed to by the field T.MBF in the terminal control block associated with the modem port.

Chapter 17 International Language Support Monitor Calls

THE LANGUAGE DEFINITION TABLE

AMOS provides for international support by implementing a language definition table for each job on the system. This table contains definitions for those items which are language dependent. For example, the characters used as a currency symbol are stored within the table, allowing a program to use whatever characters are appropriate for the currently selected language.

By providing the language definition table on a job-by-job basis, AMOS allows multiple languages to be supported on the system at the same time. In addition, any job may change its currently selected language at any time.

A default language definition table is built into AMOS at MONGEN time. Each job initially uses this default table. If properly configured, AMOS also allows each job to select the appropriate table via the SET program. Default language tables can also be selected via the MUSER program, so that the correct table is selected each time a user logs in to the system.

Contents of the Language Definition Table

The fields contained in the language definition table and their symbolic offsets are shown below. All of these offsets are defined in SYSSYM.UNV:

LD.NM1	A 20-byte field containing up to 19 ASCII characters, null terminated, that defines the name of the language. Example: "SPANISH".
LD.NM2	A 20-byte field containing up to 19 ASCII characters, null terminated, that defines an alternate name for the language. Example: "ESPANOL".
LD.EXT	A 2-byte field containing the file extension, packed RAD50, that is used for language dependent message files. For example, the message file for AlphaWRITE might be called WRTMSG, with the extension depending on the selected language. For American English, the extension is "USA," while for England it might be "UK."
LD.CSY	A 4-byte field containing up to three ASCII characters null terminated. This field defines the characters to be used as the currency symbol. Examples might be "$" for American dollars, "$A" for Australian dollars, or "Kr" for Kronar.

LD.CPS	This two-byte field defines the position of the currency symbol in relation to the currency amount. A value of zero means the symbol precedes the amount; a value of one means the symbol follows the amount.
LD.CSP	This two-byte field contains the number of spaces to output between the currency symbol and the amount.
LD.CSZ	This two-byte field contains the number of decimal digits in a currency amount. This is provided for use with decimal arithmetic for computing the maximum required precision for monetary amounts.
LD.TSP	This one-byte field contains the character to be used to separate groups of one thousand. For example, in the United States it would be a comma, while in many European countries it would be a period.
LD.DEC	This one-byte field contains the character to be used as the decimal point. In the United States it would be a period, while elsewhere it would be a comma.
LD.DTS	This one-byte field contains the character to be used as a date separator. It is usually either a "/" or a "-".
LD.DTF	A one-byte field defining the date ordering. A value of zero means "month-day-year;" one means "day-month-year;" two means "year-month-day."
LD.TMS	A one-byte field containing the character to be used as the time separator between hours, minutes, and seconds. It is usually a colon, but may sometimes be a space or a period.
LD.TMF	This one-byte field defines the time format. A value of zero means 12-hour (AM/PM) format, while a value of one means to use 24-hour (military) format.
LD.DLS	This one byte field contains the character to be used as a separator in lists of data. It is normally a comma, but this may cause problems when the period and comma are reversed from common United States usage.
LD.PPL	This one-byte field contains the character to be used as the left side of a PPN designation. It is normally a "[", but as this character is not available in all languages, it may be redefined through the use of this field.
LD.PPR	This one-byte field contains the character to be used as the right side of a PPN designation. It is normally a "]", but as this character is not available in all languages, it may be redefined through the use of this field. The left and right PPN designation characters may be the same if desired.

LD.CHS This one-byte field contains the character set number associated with this language. This character set number is intended to be used with terminals that have selectable character sets. The currently defined character set numbers are assigned as follows:

Octal Value	Hex Value	Character Set
0	0	United States (USASCII)
1	1	British (U.K.)
2	2	German
3	3	French
4	4	Swedish
5	5	Danish
6	6	Norwegian
7	7	Dutch
10	8	Italian
11	9	Spanish

LD.YES This six-byte field contains an ASCII string, null terminated, that represents the word for "YES" to be used for positive confirmation of commands. This word must consist of all upper case characters. Example: "OUI"

LD.NO This six-byte field contains an ASCII string, null terminated, that represents the word for "NO" to be used for negative confirmation of commands. This word must consist of all upper case characters. Example: "NEIN"

LD.YCH This one-byte field contains the single ASCII character to be used for a single character confirmation of commands. This character must be in upper case. Example: "Y".

LD.NCH This one-byte field contains the single ASCII character to be used for a single character negative confirmation of commands. This character must be in upper case. Example: "N".

LD.WRD This 30-byte table, terminated by a null byte, contains any characters, other than "A-Z" and "a-z," to be considered alphabetic and therefore part of a word. This table is necessary because most foreign character sets have additional letters that are placed in ASCII codes normally occupied by punctuation in USASCII. This table is used by the ALF monitor call, as well as previous word/next word type of functions.

LD.ULC This 30-byte table, terminated by a null byte, consists of 15 pairs of characters. Each pair consists of an upper case character followed by its lower case equivalent. One entry must be made in this table for each pair of alphabetic characters other than "A-Z" and "a-z". This table is used by the UCS and LCS monitor calls and any other routines that must perform case conversion. It is necessary as the additional alphabetic characters do not have a standard relationship between their upper and lower case versions.

LD.COL	This 128-byte table contains the collating sequence for the 128 printable ASCII characters. The table is intended for use in sorting operations. It is used by using the ASCII value of the character as an index into this table to obtain a collating value. This table is necessary because the additional alphabetic characters in other character sets do not have ASCII values corresponding to their desired sort sequence.
LD.JAN	This 20-byte field contains the ASCII text for the name of the month of January, terminated by a zero.
LD.FEB	This 20-byte field contains the ASCII text for the name of the month of February, terminated by a zero.
LD.MAR	This 20-byte field contains the ASCII text for the name of the month of March, terminated by a zero.
LD.APR	This 20-byte field contains the ASCII text for the name of the month of April, terminated by a zero.
LD.MAY	This 20-byte field contains the ASCII text for the name of the month of May, terminated by a zero.
LD.JUN	This 20-byte field contains the ASCII text for the name of the month of June, terminated by a zero.
LD.JUL	This 20-byte field contains the ASCII text for the name of the month of July, terminated by a zero.
LD.AUG	This 20-byte field contains the ASCII text for the name of the month of August, terminated by a zero.
LD.SEP	This 20-byte field contains the ASCII text for the name of the month of September, terminated by a zero.
LD.OCT	This 20-byte field contains the ASCII text for the name of the month of October, terminated by a zero.
LD.NOV	This 20-byte field contains the ASCII text for the name of the month of November, terminated by a zero.
LD.DCM	This 20-byte field contains the ASCII text for the name of the month of December, terminated by a zero.
LD.MON	This 20-byte field contains the ASCII text for the name of the day Monday, terminated by a zero.
LD.TUE	This 20-byte field contains the ASCII text for the name of the day Tuesday, terminated by a zero.

LD.WED	This 20-byte field contains the ASCII text for the name of the day Wednesday, terminated by a zero.
LD.THU	This 20-byte field contains the ASCII text for the name of the day Thursday, terminated by a zero.
LD.FRI	This 20-byte field contains the ASCII text for the name of the day Friday, terminated by a zero.
LD.SAT	This 20-byte field contains the ASCII text for the name of the day Saturday, terminated by a zero.
LD.SUN	This 20-byte field contains the ASCII text for the name of the day Sunday, terminated by a zero.

In addition to the fields listed above, the symbol LD.SIZ is defined as the size of a language definition table.

USING THE GTLANG MONITOR CALL

The GTLANG monitor call returns a pointer to the language definition table currently in use by your job. It is called by:

```
        GTLANG   An
```

An is any address register. You may then use this address register as an index when using the table offsets defined above.

For software which must run under versions of AMOS prior to 1.3, but wishes to take advantage of the language definition tables when they become available, you must test the SY$LNG bit in the SYSTEM word within the system communication area prior to using the GTLANG call. ***Use of the GTLANG call under AMOS 1.2A and earlier monitors may crash the system.***

DEFINING YOUR OWN LANGUAGE DEFINITION FILE

Alpha Micro provides a way for you to create your own language definition files based on your own special needs. We have supplied you with an .LDF file for American English, and a special definition file, LDFSYM.M68, which makes it easier to define your own file.

The macros used within the language definition files are all contained in the file LDFSYM.M68. Each of these macros defines one of the language definition file fields defined above. Each of these macros must be used in order, and all fields must be defined. For this reason the easiest way of creating a new language definition file is to simply modify one of the existing files.

All arguments used by the macros are decimal.

Chapter 18
Directory Handling System

AMOS supplies a set of monitor calls which perform directory access in a standard, file system independent fashion. By using these calls, you can make sure that your software will not be affected by minor changes to the AMOS file structure.

These calls function equally well on the traditional and extended format file systems. Your software can use these calls without regard to what file system is in use, greatly simplifying the task of generating directory access software.

DSKINI - INITIALIZE A LOGICAL DISK

The DSKINI call is used to initialize a logical unit by resetting the directory and bitmap areas to be completely empty. This call removes all files, including BADBLK.SYS, requiring extreme caution in its use. This call is used by the SYSACT program and will rarely be needed by user software.

DSKINI requires that the calling job be logged into DSK0:[1,2] to perform this call.

The calling sequence is:

```
        DSKINI   ddb          ; initialize the logical unit
```

Ddb references an INITed DDB referencing the logical unit to be initialized.

DSKACC - ACQUIRE DIRECTORY ACCESS

The DSKACC call is used prior to any other directory access calls to gain access to the directory and to initialize the DDB's directory marker.

The calling sequence is:

```
        DIRACC   ddb, flags      ; get access to directory
```

Ddb references an INITed DDB specifying the device whose directory you wish to access, and ***flags*** contains one or more of the following flags:

Symbol	Meaning
DA$INI	Initialize directory search from root.
DA$NEW	DDB contains a new marker: reread the directory block to make sure everything is correct.
DA$LVL	Advance to next directory level (i.e., from MFD to UFD).
DA$DRL	Lock the directory (same as DSKDRL) before doing anything else.

DIRSCH - SEARCH A DIRECTORY

This call scans through a directory, returning selected items. Flags supplied with the call allow you to specify which items or combination of items will be returned. This is the basic call used for wildcarding through directories.

The calling sequence is:

```
DIRSCH   ddb, flags      ; search thru directory
```

Ddb references an INITed DDB specifying the device whose directory you wish to search, and ***flags*** contains one or more of the following flags:

Symbol	Meaning
DS$DIR	Return directory items that are found (i.e., PPNs)
DS$DAT	Return data file items that are found
DS$DEL	Return deleted entries that are found
DS$CMP	Compare filename to DDB and return only matches
DS$INH	Inhibit transfer of directory information into DDB
DS$FNF	Set error if file not found
DS$DUP	Set error if duplicate primary file
DS$AUX	Set error if duplicate auxiliary file

After this call returns to your program, you can check the flags in D.DIR to determine which type of entry was located. These flags are:

Symbol	Meaning
DF$MFD	Current level is MFD
DF$UFD	Current level is UFD
DF$DIR	Current entry defines a directory item
DF$DAT	Current entry defines a data file item
DF$DEL	Current entry is a delete item
DF$ISF	Initial search flag (used internally)
DF$LVL	Search next level (used internally)
DF$ALC	Next level needs to be allocated (used internally)
DF$AUX	Auxiliary file was found
DF$EOD	End of directory was encountered

The DF$EOD flag will be set after all items in the current level have been searched.

DIRREP - REPLACE A DIRECTORY ENTRY

This call replaces the current directory item—the one referenced by the current directory marker—with the information contained in the specified DDB, including filename, extension, dates, times, and protection.

The calling sequence is:

```
DIRREP   ddb             ; back to AMOS
```

Ddb references an INITed DDB specifying the directory entry you wish to replace.

DIRDEL - REMOVE A DIRECTORY ENTRY

This call removes the current directory entry—the one referenced by the current directory marker—by marking it as deleted.

The calling sequence is:

```
              DIRDEL   ddb              ; back to AMOS
```

Ddb references an INITed DDB specifying the directory entry you wish to delete.

DIRALC - ALLOCATE A NEW DIRECTORY LEVEL

DIRALC allocates new directory items in an existing directory. It is intended for Alpha Micro use only.

The calling sequence is:

```
              DIRALC   ddb, flags       ; back to AMOS
```

Ddb references an INITed DDB specifying the device whose directory you wish to allocate a new entry in, and ***flags*** contains a 32-bit value equivalent to the D$TYP field described in the "Directory Block Format" section of Appendix A

SAMPLE USAGE OF DIRECTORY HANDLING CALLS

The following code fragment demonstrates the DIRACC and DIRSCH calls. It performs a wildcard search for all files with a DVR extension in DSK0:[1,6]. As each file is found, it is loaded into the caller's partition. It assumes that A5 indexes an impure area defined to contain two DDBs (SDDB and FDDB).

```
;Handle wildcard driver loading
LODDVR: LEA      A1,SDDB(A5)           ; index DDB for scanning
        MOVW     #[DSK],D.DEV(A1)      ; setup for DSK0:
        CLRW     D.DRV(A1)
        INIT     @A1                   ; get a buffer
        DIRACC   @A1,#DA$INI           ; get access to device
        MOVW     #ED.DVR,D.FIL(A1)     ; scan for [1,6]
        DIRSCH    @A1,#DS$DIR+DS$CMP
        DIRACC    @A1,#DA$NEW+DA$LVL   ; drop down a level to files

;Loop here for each data file in the directory
110$:   DIRSCH   @A1,#DS$DAT           ; get another data file
        TSTW     D6                    ; end of files?
        BMI      120$                  ;   yes -
        CMPW     D.EXT(A1),#[DVR]      ; is this a device driver?
        BNE      110$                  ;   no -
        CLEAR   FDDB(A5),D.DDB         ; start with fresh DDB
        MOV      D.FIL(A1),FDDB+D.FIL(A5)      ; set filename
        MOVW     D.EXT(A1),FDDB+D.EXT(A5)      ;   and extension
        FETCH    FDDB(A5)                      ; load the driver
        BEQ      110$                          ;   and go get more
        TYPECR   <?Unable to fetch driver>

;All done –- return to AMOS
120$:   EXIT
```

Chapter 19
System Disk Cache Calls

AMOS provides an integral disk cache buffer system as part of its file service system. By buffer frequently used disk blocks in memory, AMOS is able to substantially reduce the amount of physical disk IO performed, substantially improving overall system response speed.

While this buffering process is normally completely transparent to all software, there are times when a program needs to gain access to some of the control parameters used by the disk cache system. By properly manipulating these controls, system performance can be tuned for a particular application.

For more information on the disk cache system, and how the various parameters can be used, see the section on the Disk Cache Buffer Manager in the *AMOS System Operator's Guide*.

STRUCTURE OF CALLS TO THE DISK CACHE SYSTEM

If you wish to write assembly language routines that use the disk cache system, the following sections describe the various calls you can make. Each of these calls is made by calling the disk cache system through a special dispatch vector (DCACHE) in the System Communications Area, rather than via the more usual monitor calls.

Before making one of these calls, you must load the address of a DDB into register A4. The exact contents of the DDB will depend on the call being made.

The sequence used for these calls is:

```
MOV      call function code,D7
MOV       DCACHE,A6
CALL      @A6
```

Cache Function Codes

Calls to the disk cache manager require a function code in register D7 to identify the type of call being made. The available function codes are:

Symbol	Octal Value	Hex Value	Purpose
DC.LM	1	1	Lock MFD
DC.LU	2	2	Lock UFD
DC.LF	3	3	Lock a file
DC.LB	4	4	Lock a block
DC.UM	5	5	Unlock MFD
DC.UU	6	6	Unlock UFD
DC.UF	7	7	Unlock a file
DC.UB	10	8	Unlock a block
DC.CM	11	9	Clear MFD
DC.CU	12	A	Clear UFD
DC.CF	13	B	Clear a file
DC.CB	14	C	Clear a block
DC.CD	15	D	Clear a unit
DC.ON	16	E	Turn on the cache
DC.OF	17	F	Turn off the cache
DC.DM	20	10	Dynamically lock MFD
DC.DU	21	11	Dynamically lock UFD
DC.XM	22	12	Dynamically unlock MFD
DC.XU	23	13	Dynamically unlock UFD

For example:

```
MOV     #DC.LF,D7
MOV     DCACHE,A6
CALL    @A6
```

Error Codes

All function calls, except the ON and OFF calls, perform error checking and report an error status on return. When an error is detected, the error status byte in the DDB is set to the appropriate error code. Also, on return, the Z-bit is set to indicate a good completion or cleared to indicate an error. There is no option to abort on error or to print an error message.

Error codes that are less than 200 (octal) indicate a monitor error and codes greater than or equal to 200 (octal) indicate a cache error. Currently, the only cache error code reported by the cache manager is ER.NOS (Not enough cache space available). The remaining cache error codes are used elsewhere. The error codes are:

Symbol	Octal Value	Hex Value	Meaning
ER.SPC	201	81	Specification error
ER.CMD	202	82	Command error
ER.ARG	203	83	Argument error
ER.SWX	204	84	Switch error
ER.NEX	205	85	Disk cache does not exist
ER.INA	206	86	Disk cache is off
ER.ONX	207	87	Disk cache is already on
ER.OFF	210	88	Disk cache is already off
ER.NOS	211	89	Not enough cache space available
ER.LOC	212	8A	Locked
ER.UNL	213	8B	Unlocked
ER.CLR	214	8C	Cleared

Chapter 20
AlphaTCP Programming Interface

AlphaTCP is Alpha Microsystems' implementation of the industry standard DARPA TCP/IP family of protocols. AlphaTCP allows your AMOS applications to communicate with applications running on a variety of systems.

The AlphaTCP TAME interface (TCP Access Made Easy) is an event-driven monitor call interface for AlphaTCP. TAME transparently handles the complexities involved with translating host names, handling a connection in a non-blocking mode, and a variety of other functions.

COMPATIBILITY

TAME requires AlphaTCP 1.4 or later, and AMOS 2.3 or later. It is not compatible with the 1.X versions of AMOS. The AlphaTCP TAME server must be active on the system. See the *AlphaTCP Administrator's Guide* for information on setting up servers.

Only the TCP protocol is supported, UDP is not available through the TAME interface.

TCP PROGRAMMING OVERVIEW

The next sections provide a brief overview of TCP application programming. You should be familiar with these concepts before you use the TAME interface.

Client/Server Paradigm

The majority of TCP/IP software is written according to the client/server paradigm. The concept is that *server* programs provide valuable services and functions to *client* programs, which will contact them as needed.

Servers usually start at boot time and never exit. They passively listen in the background until a request is made of them; they do not normally have a terminal attached. Because they must be contacted, they listen on *well known port numbers*. If you imagine the U. S. telephone system, servers will always be contacted on a well known number (such as *0* or *911)*, which can also be associated with a well-known service name (*operator* or *emergency*). Port numbers and service names are explained below. In the UNIX world, servers are often referred to as *daemons*.

Clients usually contact a server to perform their operations. Since they do not usually need to be contacted themselves, they may initiate their conversation on whatever port the system assigns them at the time. This is known as an *ephemeral port*. Again, if you imagine the U. S. telephone system, it does not matter which telephone number you are calling *from*. (There might be an argument for the emergency number 911, but just like 911 a server *can* identify where the communication originated.)

IP Addresses and Port Numbers

A TCP/IP communication endpoint is defined by a *protocol*, an *IP address*, and a *port number*. A virtual communication channel exists between two applications when each application's endpoint references the other application's endpoint using a common protocol. This association is known as a *5-tuple*, because it contains two IP addresses, two port numbers, and one protocol:

- The *protocol* is the method used to communicate between applications via a physical interface (i.e., the Ethernet). It is the language spoken across the interface, much like speaking English or French during a telephone conversation. Just as both sides of this conversation must speak the same language, both sides of a TCP/IP connection must speak the same protocol. The most commonly used protocols are *TCP* and *UDP*. TAME supports TCP only.

  TCP is a reliable protocol which handles the intricacies of data delivery over potentially unreliable paths. At the same time, it adjusts to the varying bandwidth available when networks are connected using slower links.

- TCP/IP uses the *IP address* to define a system. The IP address is often shown as four decimal values separated by periods. Each of these values may be from 0 to 255. The IP address is comparable to a telephone number.

  An example of an IP address is *127.0.0.1*. This IP address is the loopback address: the address that points back to your local system so network applications can communicate without a network.

- The *port number* identifies a place on the system where data may be read or written. Often, this means a running application. Any application may have many communication channels open. Port numbers range from 1 to 65535.

  Port numbers under 1024 often have a special meaning since certain values are assigned to standard applications. These are the *well-known* port numbers. Most systems begin allocating temporary port numbers at 1024. Numbers under 1024 were considered more secure due to a feature in UNIX; however, with the proliferation of PCs ,this characteristic can no longer be taken for granted.

An actual conversation is defined by the complete 5-tuple. Thus, it is valid to see many separate conversations active using the same port numbers, as long as some portion of the association is unique.

Service Names

Service names are given to the well-known ports to make them easier to remember. Whenever TCP/IP applications encounter a service name, they look it up to get the port number associated with it. When programming, use service names, choosing to connect to a *service* or provide a *service.* The system will handle the actual *port number*. For example, the *telnet* service uses the well known port number *23.* Examine the file TCP:SERVIC. for the services commonly used with AlphaTCP.

Host and Domain Names

Systems are assigned names in a hierarchy of *host name* and *domain name*. The full specification of host name with domain name is known as the systems *fully qualified domain name* (FQDN). Whenever a TCP/IP application encounters a name, it looks it up to find the real IP address associated with it. If the name is not fully qualified, portions of the domain name are appended to find a match.

Much like IP addresses, the FQDN is divided into sections separated by periods. Unlike IP addresses, there is no fixed number of sections. This allows for flexible naming conventions which can fit the organizational structure of a site. The domain name may contain many sections itself. From left to right, sections of the FQDN represent higher levels of administration, with as many levels as an organization needs.

For example, the name *orders.widgit.com* can represent the order entry system at Widgit, Inc.'s headquarters. Taking this further, *pc.tx.widgit.com* can represent the Production Control system at their plant in Texas, while *mail.tx.widgit.com* can represent the mail system at that same plant.

The *com* section is a virtual *root domain* that represents commercial entities. A university would use *edu* instead. A different style is used for international sites. More information about root domains can be found in the *AlphaTCP Administrators Guide*, or most other TCP/IP publications.

TAME PROGRAMMING OVERVIEW

Following is an overview of the programming implementation presented by the TAME API.

Procedural Versus Event-Driven

In use today are two primary models of software development, procedural and event-driven. TAME uses the event-driven model, which is a newer style popularized by the advent of GUI operating systems.

The procedural model follows a step-by-step flow of control. Most procedural TCP/IP programming is written with *blocking synchronous* calls. In other words, the call will not return until satisfied. For example, a procedural server writes out a welcome message and waits for the client to issue its request. If other writes were done previously, the welcome message write could pause. Nothing can be done until the you provide input, and then the program performs its next step. You must address error conditions in every section of the program. Modifying the behavior of a procedural program can be difficult. Procedural programs tend to interact with their environment in many different places in the code, the execution of which depends on past operations.

The event-driven model follows no strict flow. The program reacts to stimuli known as *events*. Because events may come in at different times in different order, most event-driven TCP/IP programming is written with *non-blocking asynchronous* calls. In other words, the calls always return right away. Some calls simply initiate an operation, while others will return an error if they were called before they should have been. You are notified of call completion and readiness through processing of events. In the previous example, you will know when it's OK to write without pausing through an event. You will know the same way when input is available. Your application is free to perform other tasks in between.

Event-driven programs tend to have a main loop of execution where interaction with their environment takes place, and state flags to control behavior from past operations.

The TAME interface supports the event-driven model of programming. All TAME operations are non-blocking and so, all the monitor calls will return right away without waiting for completion. You are notified of important events through software interrupts, thus your application is *event-driven*. Examples of events include the presence of input, the ability to accept more output, and connection establishment or shutdown.

TAME's use of AMOS software interrupts is an important aspect of its event-driven nature. Software interrupts cannot be processed while in another wait state, for example the *terminal input* wait state. Your entire application should be event driven, or at least written to use the TCPWAT or SIWAIT monitor calls. You should avoid calls which will place you in a non-event-driven style wait state. If you are not familiar with software interrupts, refer to chapters 14 and 15 in this manual. There are also a number of books available on the basics of event-driven programming.

Features

AMOS monitor calls provide access to TAME. TAME provides a large number of calls useful in network programming, plus several support functions. Monitor calls are provided to:

- Establish, accept, and close network connections.
- Read and write data.
- Transfer control of a connection to another job.
- Return useful connection information.
- Dynamically spawn other jobs.
- Pass information between jobs.
- Convert path and filenames to AMOS format in a standard way.

Program Operation

Clients and servers usually differ in operation only at the start and end of a connection. As a client, an application will first attempt to establish a connection. As a server, an application will begin listening for connections, then accept the connections as they are requested. After a connection is terminated, a client will usually terminate while a server listens for and processes new connections.

Once a connection is established, operation of a client or a server is pretty much the same. The application checks for events, retrieves the active events and services them as needed. Then, it notifies TAME when it has completed processing events. This cycle is repeated until the connection is not needed any more, and the connection is closed.

Client Operation

A client establishes a connection with a server. Upon successful connection, data will be read and written. After completing the transaction, the client may opt to close the connection or wait for the server to do it. This depends on the design of the application. After this, a client usually exits.

When designing both the client and server applications, it is best to let the client close the connection first. The reason is a TCP state known as TIME-WAIT. The end *initiating* the close must keep information about the connection for one to four minutes after closing the connection. This ties up system resources. Since clients often run on many systems while the server runs on one, allowing the clients to initiate the close spreads the TIME-WAIT support between more systems. If you design your application to have the server close first, all TIME-WAIT resources will be concentrated on the server system.

Server Operation

A server listens to a well-known port number. When notified of a client connection, it accepts the connection. This creates a new handle, allowing the original handle to keep listening.

Once a server has accepted the connection, it may chose to do one of two things. It may process the connection itself through events, or it may spawn another server to handle each connection. This is purely a design decision and depends on your application's requirements.

Servers which don't spawn will allocate a control structure for each new connection. It then processes and responds to events as needed, cleaning up the control structure upon termination of the connection. You must avoid lengthy processing on any one connection.

Servers which spawn pass the connection handle to the spawned child, possibly on the command line or through the *attention event* mechanism. Upon starting, the child should request control of the session by using the passed handle, thus causing a handoff event in the parent. The parent should only process the *handoff* event on the connection handle, any other events should simply be acknowledged. Once the parent has relinquished control, the child can then use the new handle returned to control the session.

Making Requests

Through monitor calls, connections are established and terminated, and data is read and/or written. Every one of these monitor calls will return without a significant delay, though the operation requested may take time to complete.

For example, the process of establishing a connection is simple and returns quickly;

```
        TCPCNI D6,HANDLE(A5),<dev.widgit.com>,<pop3>
```

The underlying mechanism, however, is complex. The service name *pop3* must be converted to a port number and the hostname *dev.widgit.com* must be converted to an address. This may involve contacting several other systems known as *name servers*. The host itself must acknowledge acceptance of the connection. Only then is the connection request actually complete. This may take less than a second, or several minutes. There are also a number of places which can fail along the way. In a later section we discuss *events* that notify you of completion and failure.

TAME connections are called *sessions*, whether completed or not. To avoid the problems associated with stale memory addresses, all sessions are addressed using a unique *handle*. The handle is simply a four-byte value provided by TAME which will not be reused for a long time, thus providing an identification mechanism.

The monitor calls used to make requests are listed below. Each one is discussed in detail later in the text. Note that some of these calls will return while the action is pending. Other calls cannot be made unless the connection is prepared for them and will return an error. These requirements are supported by proper handling of the various *event* notifications you receive during the life of a session.

Handling Events

Events provide support for the asynchronous non-blocking nature of TAME. *Software interrupts* notify the program of events. While there are several different software interrupts available, TAME uses the *SI$TCP* software interrupt for event notification. Software interrupts are described in chapters 14 and 15 of this manual.

Events are serviced, or flagged as needing service, within the SI$TCP software interrupt handler in your application. The handler will query TAME for current events using the TCPEVT or TCPPOL monitor calls, which return event flags in a *word* size field. Upon completion of all event servicing, the TCPEDN monitor call notifies TAME that event servicing has completed, thus allowing a new cycle.

Certain data movement events are handled directly within TCPEVT or TCPPOL themselves. Because of this, it is important to perform TCPEDN *only* when events are reported to you. If no events were reported to you, it's possible the only events pending were the hidden data movement ones, thus no TCPEDN is needed. In other words, it is possible to receive a software interrupt indicating events exist, only to query for those events and have nothing reported. In this case, the only events which existed were the hidden data movement ones and those were already serviced by the polling.

There are two styles of event:

- Global events, which are not directly related to any session.
- Session events, which are unique to each session.

The events you may see are listed below. Each one is discussed in detail further on.

MONITOR CALL SUMMARY

TAME provides a large set of monitor calls. While only a handful of these are required for most TCP/IP programming, TAME has been developed to allow for a variety of complex client/server implementations.

Avoid using data registers with any of the monitor calls; many of them are used internally by the calls for passing on data.

Client Side Active Connection Establishment

- TCPCON—Connect using indexed strings

- TCPCNI—Connect using immediate strings

Server Side Passive Connection Acceptance

- TCPLSN—Listen using indexed strings
- TCPLNI—Listen using immediate strings
- TCPACC—Accept an incoming connection request

Connection Shutdown

- TCPDSC—Terminate active or passive session

Data Movement

- TCPRED—Read currently available data (amount *will* vary)
- TCPFIL—Read in a data record of fixed size (begin record transfer)
- TCPLIN—Read in a line of data (begin line transfer)
- TCPWRT—Write data

Event Processing

- TCPEVT—Return global and session events
- TCPPOL—Return global and session events for all sessions
- TCPEDN—Completed event servicing, ready for more

Spawned Job Support

- TCPSPN—Spawn a job
- TCPDES—Cleanup terminated spawned jobs
- TCPKIL—Terminate a spawned job
- TCPREQ—Request control of a session (for a spawned job)
- TCPREL—Relinquish control of a session (for parent job)

Other Calls

- TCPPID—Return process ID information
- TCPSAT—Set an attention event
- TCPQAT—Query for attention event information
- TCPINF—Return connection information for session
- TCPCPF—Convert a path specification to an AMOS filename
- TCPWAT—Preemptively wait for a software interrupt
- TCPRES—Resolve a hostname to address, or address to hostname

EVENT SUMMARY

Global Events

- TG$CHILD—A spawned job has exited (parent job only)
- TG$PARENT—Parent of spawned job has exited (spawned job only)
- TG$ATTENTION—General signaling method for parent and spawned jobs
- TG$DOWN—The TAME interface is being shut down

Session Events

- TE$CONNECTED—Connection established
- TE$DISCONNECTED—Connection closed by remote
- TE$DROPPED—Connection abruptly terminated or failed establishment
- TE$DATAIN—Unspecified amount of data ready to be read
- TE$DATAOUT—Interface ready for sending data
- TE$RECORD—Record fill or line read has completed (data ready)
- TE$HANDOFF—Control of a session has been requested
- TE$TRUNCATED—Connection closed with a partially filled line or record
- TE$RESOLVED—Name or address query has completed

THE EVENT HANDLER

The event handler is the heart of an application using TAME. While the rest of the application may request operations, it is the event handler which controls scheduling and completion of the requests. For instance, if the event handler has not been notified that a connection can accept data, a write attempt by the application will fail.

You may query a session for events and find no events pending; if so, you should <u>not</u> automatically perform TCPEDN. To ensure fair sharing of processor time, TAME performs certain data movement operations with special events. These special events are serviced directly within the TCPEVT call itself, which will perform its own TCPEDN unless more events are pending to be serviced by your application.

Setting Up

To develop a true event-driven application, you must use a variety of software interrupts. For more information on the use of software interrupts, see chapters 14 and 15 in this manual. The *SI$TCP* software interrupt handles event notification.

To set up the event handler:

1. Define a software interrupt vector table in your impure area, for example:

   ```
   .OFDEF    SITBL,SI.SIZ
   ```

2. You need a mechanism to pass the impure pointer, since software interrupt code cannot rely on the state of any registers. For a stand-alone program, use the job's DSECT pointer in the JCB. The example below assumes the impure area is indexed by A5:

   ```
   JOBIDX
   MOV      A5,JOBDSC(A6)
   ```

3. Place a reference to your event handler code within the table, register the table, and enable the event handler. The example below assumes the impure memory is indexed by A5:

   ```
   LEA      A6,TCP.EVENT
   MOV      A6,SITBL+SI.TCP(A5)
   SIMSK    SITBL(A5),#SI$TCP
   ```

4. You will use other software interrupts when developing a truly event-driven application. Place references to all these event handlers in the vector table prior to the SIMSK above.

5. Enable them all at once by OR-ing together all of the mask flags on a single SIMSK line.

Accepting Events

The event handler code is called with the same job context as the main program. It saves all modified registers, and may not rely on the existing content of any of them. Be aware that most monitor calls change A6, D6, and D7 ***even if you don't***. The event handler should index the programs impure area, then determine which events need servicing. If you follow the examples above, the start of the event handler may look like the following assuming there is an entry in the impure area called *HANDLE*, into which a prior connect or listen call stored a returned session handle. Further, assuming there are a pair of *words* where we put global and session events:

```
TCP.EVENT:
        SAVE     A5,A6,D6,D7
        SICLR    SI$TCP
        JOBIDX
        MOV      JOBDSC(A6),A5
        TCPEVT   D6,HANDLE(A5),GEVENTS(A5),SEVENTS(A5)
        BNE      BAD.HANDLE
        REST     A5,A6,D6,D7
        SIRTN
```

The code above resets the active SI$TCP software interrupt. It then retrieves the impure memory reference from the job's DSECT vector. Finally, it requests the pending events from the TAME subsystem. If an error occurs, the offset BAD.HANDLE may interrogate D6 and act appropriately.

There are many ways to write the event handler. It may service the events itself, or it may simply store them for processing in the main body of the program. The example above stores the event for later processing. No further events will arrive until a TCPEDN is performed, and executing TCPEVT again will simply return the same events.

Note the use of SIRTN instead of RTN in the example above. SIRTN is the proper instruction for leaving a software interrupt routine.

Servicing Events

The example above stores events to be processed within the main body of the program. To keep your program from using CPU time when there is nothing to do, use the TCPWAT call. For example:

```
MAIN.LOOP:
        CTRLC    EXIT.PROGRAM
        SUPVR
        TSTW     GEVENTS(A5)
        BNE      10$
        TSTW     SEVENTS(A5)
        BNE      10$
        TCPWAT
        LSTS     #0
        BR       MAIN.LOOP
10$:    LSTS     #0
        CALL     EVENT.PROCESSING
        BR       MAIN.LOOP
```

The SUPVR and LSTS #0 statements are used to avoid a deadlock condition. Software interrupts are normally processed during monitor calls and context switches, but not if the job is in supervisor mode. By using supervisor mode, the possibility of GEVENTS or SEVENTS becoming set after the test but before the TCPWAT is eliminated. CTRLC is used outside of supervisor mode not only to check for ^C, but to process any outstanding software interrupts. The JOBIDX monitor call is another fast monitor call which may be used to force processing of software interrupts.

The EVENT.PROCESSING subroutine calls appropriate code for each set event. When all pending events are processed, the EVENT.PROCESSING subroutine clears the event flags and performs the following operation:

```
        TCPEDN   D6,HANDLE(A5)
```

This tells TAME that the program is ready for new events. To tell tame you are finished processing global events should any occur, use the following operation:

```
        TCPEDN   D6,#0
```

Keep an eye out for the following conditions when building a TAME program:

- Never perform a TCPEDN if the TCPEVT call does not return any active events. Certain special data movement events are serviced within TCPEVT. If the only events present were the special data movement events, TCPEVT performs its own TCPEDN and returns with no events pending.
- When addressing global events or multiple sessions, it is possible to get deadlocks or missed SI$TCP signals. When you are expecting events from multiple sources (i.e., multiple sessions or spawned children), insure that you provide interlocking. Process the events directly in the event handler, where the SI$TCP software interrupt is blocked pending the SIRTN. A more complex approach involves the use of flags.
- In an event-driven program , it is important to avoid extended wait states, other than software interrupt wait (Si). One example is terminal input wait (Ti). If you enter a non-event-driven wait state, the program will be unable to process software interrupts. To avoid using the terminal input wait state, use TCPWAT or SIWAIT instead, which will be awakened by keyboard input as well

as other events. You may then choose to register a keyboard handler, or simply use TCKI prior to retrieving keyboard input. Use either method to avoid blocking in a non-event-driven wait state.

MONITOR CALLS

The following section details all the monitor calls. It is important to remember that:

- Monitor calls always destroy registers A6, D6, and D7.
- Parameters are passed internally in data registers.
- A6 is used to get indexes to memory-based arguments.

Although monitor calls save any registers that get modified (except the ones mentioned above), it's not a good idea to use a data register in an argument since the order used internally might conflict with a subsequent argument using it (i.e. if the call uses D2 internally to reference argument #1, then passing 0(A0)[D2] as argument #2 would result in a bad value for argument #2). Examining the structure of the TAME macros will give you a better understanding. As error conditions are returned in D6, it is safe to use D6 as the error argument.

Establish a Connection

To establish a connection to a server, use either:

```
TCPCON  error, handle, host, port, {sourceport}, {flags}

TCPCNI  error, handle, host, port, {sourceport}, {flags}
```

TCPCON uses indexed references to host and port, while TCPCNI uses immediate strings on the line. Use either of these two calls to establish a connection to a server on ***host*** which is listening on ***port***. The *port* argument may be a numerical string, or a *service name* defined in TCP:SERVIC. The call returns ***handle***, which is a longword used to reference the connection in other calls. The argument ***sourceport*** is optional and may be a numerical string or service name. If the source port is not specified the next available port will be used.

The following options may be specified for the optional argument ***flags***:

Symbol	Meaning
TC$KEEPALIVE	Enables keepalives for clients.
TC$NODELAY	Disables NAGLE small-packet delay.
TC$FIRSTADDR	Only attempt first IP address.

Keepalives are mechanisms transparent to the application which periodically test an idle connection. If the remote does not respond for an extended period (by default two hours), the connection is reset. If the remote has been rebooted, the connection will be reset at the next keepalive (two minutes). Keepalive is used to prevent a server from waiting in the background endlessly and is OFF for active connects (implied clients) by default. Specifying TC$KEEPALIVE turns it ON.

NAGLE is a TCP feature that attempts to queue up tiny data fragments into larger ones, thus lowering the overall packet overhead. This is very important on slower connections, or busy or long routes, since TCP packet headers are usually 40 bytes in length. NAGLE works by allowing only 1 unacknowledged packet in transit at a time. On a fast Ethernet, NAGLE is transparent, but on serial connections, it groups things like keystrokes into single packets for more efficient transmission. By default NAGLE is enabled. Specifying TC$NODELAY turns it OFF.

A name lookup could return multiple IP addresses for a single host, when using domain name service. This happens when a host contains multiple network interfaces. By default, if a connection cannot be established on one address, TCPCON and TCPCNI try the next one. When you specify TC$FIRSTADDR, it only tries the first address. This switch should not be used in most situations.

Examples of the above calls:

```
TCPCON   D6,HANDLE(A5),HOST(A5),PORT(A5)
TCPCNI   D6,HANDLE(A5),<dev.widgit.com>,<pop3>,,#TC$KEEPALIVE
TCPCNI   D6,HANDLE(A5),<test.widgit.com>,<2001>,<2002>
```

Listen for Connections

To listen for connection requests on a specific port, use either:

```
TCPLSN   error, handle, port, {flags}

TCPLNI   error, handle, port, {flags}
```

The ***port*** argument may be a numerical string, or a *service name* defined in TCP:SERVIC. The call returns ***handle***, which is a longword that references the listening session. Incoming connections generate TE$CONNECTED events, however the listening session is never used to transfer data. Instead, when a connection arrives the TCPACC call should be used to assign a new handle for the active connection.

TCPLSN uses indexed references to host and port, while TCPLNI uses immediate strings on the line. The following is a list of alternatives to use for the optional argument ***flags***:

Symbol	Meaning
TL$GETNAME	lookup the connecting hostname.
TL$FAILNAME	Ignore connection if lookup fails.
TL$NOKEEPALIVE	Disables keepalives for servers.
TL$NODELAY	Disables NAGLE small-packet delay.

By default, TCP/IP accepts an incoming connection as-is (with only the source address and port number). Use TL$GETNAME to look up the source hostname prior to notifying the application of the connection.

To drop any connections for which the name is unavailable, use TL$FAILNAME. The application is never notified of the connection attempt. TL$GETNAME is automatically implied by this option.

Remember that using the above flags delays establishing the connection. Also, the name may not always be available for various reasons, which will cause another delay or prevent the connection.

Keepalives are mechanisms which transparently trigger periodical testing of an idle connection. If the remote does not respond for an extended period (by default two hours), the connection is reset. If the remote has been rebooted, the connection will reset at the next keepalive (two minutes). Keepalives are used to prevent a server from waiting in the background endlessly. The default setting is ON for passive listens (implied servers); specifying TL$NOKEEPALIVE turns it OFF.

NAGLE is a TCP feature that queues up tiny data fragments into larger ones for transmission, thus lowering the overall packet overhead. This is very important on slower connections, or busy or long routes, since TCP packet headers are usually 40 bytes in length. NAGLE allows only 1 unacknowledged packet in transit at a time. On a fast Ethernet, NAGLE is transparent, but on serial connections, it groups things like keystrokes into single packets to expedite transmission. By default NAGLE is enabled. Specifying TC$NODELAY turns it OFF.

Following are examples of the calls discussed above:

```
TCPLSN   D6,HANDLE(A5),PORT(A5)
TCPLNI   D6,HANDLE(A5),<pop3>,#TL$GETNAME
TCPLNI   D6,HANDLE(A5),<2001>,#TL$GETNAME+TL$FAILNAME
```

Accept a Connection

To accept a connection on a listening handle, use:

```
TCPACC   error, oldhandle, newhandle
```

Once a server receives a TE$CONNECTED event on the listening session ***oldhandle***, the connection is accepted and assigned its own session using TCPACC. The handle for the new session is returned in ***newhandle***. The connection is processed using the new handle, while the listening session may receive further connections on oldhandle. To accept a connection:

```
TCPACC   D6,HLISTN(A5),HACTIV(A5)
```

Terminate a Session

To disconnect or terminate a session, use:

```
TCPDSC   error, handle, reset
```

TCPDSC is used to disconnect an active connection, or otherwise terminate a session referenced by ***handle***. Upon return, handle will be invalid and should no longer be used. The ***reset*** argument is optional. When it is used, it discards any data waiting to be sent. If reset is not used, it continues to deliver any data waiting to be sent. The value of reset does not matter, only its presence on the line. For example:

```
TCPDSC   D6,HANDLE(A5)
TCPDSC   D6,HANDLE(A5),#1
```

Read Data Stream

To read data transmitted by a session into a buffer, use:

```
TCPRED   error, handle, buf, maxlen, len
```

TCPRED reads incoming data from the session referenced by ***handle*** into the buffer indexed by ***buf***. A maximum of ***maxlen*** characters will be read. The actual number of characters read is written to ***len***.

Please note that the number of characters read will not always match the number written by individual writes on the remote end. The TCP protocol can adjust the size to make a read return partial or multiple sections, as needed. For fixed record sizes, you must devise a mechanism yourself, or use the TCPFIL monitor call instead.

Example of a read data call:

```
TCPRED   D6,HANDLE(A5),BUFFER(A5),#BUFLEN,READSZ(A5)
```

Read Data Record

To read fixed amounts of data from a session:

```
TCPFIL   error, handle, buf, reclen
```

TCPFIL launches the reading of fixed amounts of data from the session referenced by ***handle***. Data is read into the buffer indexed by ***buf*** until ***reclen*** characters are read. This is a fully event-driven call. Data is transferred by your program transparently during TCPEVT calls. Once the buffer is filled in an event is triggered notifying you of completion of the TCPFIL call. To read another record you must perform another TCPFIL call. A good place to do this is in response to TE$DATAIN events. In other words, when you receive a TE$DATAIN event, perform the TCPFIL call rather than reading data. The TCPFIL call is used as follows:

```
TCPFIL   D6,HANDLE(A5),RECORD(A5),#RECSIZ
```

If the connection is closed prior to the record being completed, a TE$TRUNCATED event will occur. The number of characters completed may then be found using TCPINF.

Read a Line

Use TCPLIN to read a line of data:

```
TCPLIN   error, handle, buf, maxlen
```

TCPLIN launches the reading of a line of data from the session referenced by ***handle***. Data is read into the buffer indexed by ***buf*** until a linefeed is encountered, or ***maxlen***-2 characters are read. If you don't pair a linefeed with a carriage return, one will be added before the linefeed. In all cases, the returned string will be terminated with a null.

This is a fully event-driven call. Data is transferred by your program transparently during TCPEVT calls. Once the line is complete an event is triggered notifying you of completion of the TCPLIN call. To read another line you must perform another TCPLIN call. A good place to do this is in response to TE$DATAIN events. In other words, when you receive a TE$DATAIN event, perform the TCPLIN call rather than reading data. The TCPLIN call is used as follows:

```
TCPLIN   D6,HANDLE(A5),STRING(A5),#STRLEN
```

If the connection is closed prior to the line being completed, a TE$TRUNCATED event will occur. The number of characters completed may then be found using TCPINF.

Write Data

To write a fixed number of characters, use:

```
            TCPWRT   error, handle, buf, len
```

TCPWRT writes ***len*** characters indexed by ***buf*** to the session referenced by ***handle***. This call should be used only once per event, and only after an event indicates it is OK to write data. The data buffer should also remain unchanged until another write event occurs. To maintain the non-blocking nature of this call, data is actually moved during the TCPEVT call as resources permit. A new TE$DATAOUT event indicates that it is OK to write data. It also indicates that any previous write buffer used is free and can be reused.

The buffer sizes in the stack allow a maximum of 2,048 bytes of data to be written in a single TCPWRT call. Larger writes require additional TE$DATAOUT events and TCPWRT calls. Since TCP connections are byte streams without record boundaries, this is not a limitation. To write larger amounts of data, limit individual writes to around 1,024 bytes. This fits well in an Ethernet frame, thus the stack won't have to fragment and reassemble the data to make it fit. Since writes larger than 1,024 bytes are placed in 2,048 byte buffers, it also avoids waste and frees up more large buffers for storing incoming Ethernet packets.

Examples of the above call:

```
            TCPWRT   D6,HANDLE(A5),WRTBUF(A5),WRTLEN(A5)
            TCPWRT   D6,HANDLE(A5),ASCLBL,#ASCEND-ASCLBL
```

Query Events

To identify pending current events for a session, use:

```
            TCPEVT   error, handle, gevents, sevents
```

TCPEVT requests current events pending for the session referenced by ***handle***. Active events for the session set bits in the word **s*events***. Global events set bits in the word ***gevents***.

If you are only interested in the global events, pass a handle number of zero and only the *gevents* argument will be filled in.

Even if you are not requesting session event information, you still need to pass a *sevents* argument to the call.

Examples:

```
            TCPEVT   D6,HANDLE(A5),GEVENTS(A5),SEVENTS(A5)
            TCPEVT   D6,#0,GEVENTS(A5),D7
```

Query Multiple Events

To identify pending current events for all sessions owned by a specific job, use:

```
            TCPPOL   error, gevents, array, max, filled
```

TCPPOL requests current events pending for all sessions owned by the job performing the TCPPOL call. Only those sessions with active events will be returned. The ***array*** argument indexes an array of longword/word pairs. The array will be filled in with (longword) handle and (word) events pairs, up to a maximum of ***max*** pairs. The number of entries filled in is returned in the longword ***filled***. Global events set bits in the word ***gevents***.

Each session listed in the array should be processed individually, with a TCPEDN performed on each handle in the array that has active events.

Example of the above call:

```
TCPPOL   D6,GEVENTS(A5),EVNARR(A5),#ARRMAX,EVNCNT(A5)
```

Completing Event Processing

Use this call to indicate that all events for the session are complete:

```
TCPEDN   error, handle
```

TCPEDN notifies the TAME subsystem that processing of all current events is complete for the session referenced by ***handle***. This frees the application to be notified of further events. When dealing with multiple sessions, you will have to address a number of issues to avoid signal loss and deadlock conditions.

If global events have been processed, pass a handle number of zero. If no events were returned for a session, do not perform this call on the session. It's possible the only events present were the transparent data movement ones processed in the TCPEVT and TCPPOL calls.

Examples of the above call:

```
TCPEDN   D6,HANDLE(A5)
TCPEDN   D6,#0
```

Getting Process ID Information

Use this call to return job and parent process IDs:

```
TCPPID   error, pid, ppid
```

TCPPID returns the job's process ID, and the job's parent process ID if applicable. Process IDs are the method used by AlphaTCP and the TAME interface to identify and control jobs using TCP/IP. Upon return, the longwords ***pid*** and ***ppid*** are filled in. If the job was not spawned by a TAME-aware application, then ppid is set to 0; otherwise it contains the process ID of the job which spawned it.

Note that this will be a different ppid than listed by the PS program. For control reasons, TAMED is always the parent process listed by PS.

Setting an Attention Event

Use this call to set an attention event for a specific process ID:

```
TCPSAT   error, pid, arg1, arg2
```

TCPSAT sets an attention event for the process in ***pid***. The longword arguments ***arg1*** and ***arg2*** are passed to the process and are user defined. As the actual processing of the event in the other job is asynchronous, it's best not to pass pointers to memory areas, though memory based arguments are encouraged. For example, you can use the form ARG(A5) as long as ARG(A5) makes no reference to another memory location. Arguments not provided default to 0. The attention mechanism is handy for passing control of a session to a child.

Querying for Attention Event Information

Use this call to return information for an attention event:

```
TCPQAT   error, pid, arg1, arg2
```

TCPQAT writes the signaling process ID to ***pid*** and the arguments passed with the attention call are written to the longwords ***arg1*** and ***arg2***. Arguments not provided default to 0.

Getting Connection Information

Use this call to return information about a specific connection:

```
TCPINF   error, handle, info
```

TCPINF returns information about a call referenced by ***handle***. The information is returned in the structure referenced by ***info***. TCPINF returns local and remote addresses and port numbers. The local hostname is returned as found in TCP:MYNAME.. Also, to return the remote hostname for incoming connections, configure the TCPLSN call to look it up with TL$GETNAME. Furthermore, TCPINF returns a distance indicator in relation to the remote host.

The format of the ***info*** structure is as follows:

Symbol	Size	Purpose
tai.lname	4	Pointer to local name string storage
tai.lnsiz	4	Size of above string
tai.laddr	4	Pointer to local address string storage
tai.lasiz	4	Size of above string
tai.lport	2	Local port number
tai.rname	4	Pointer to remote name string storage
tai.rnsiz	4	Size of above string
tai.raddr	4	Pointer to remote address string storage
tai.rasiz	4	Size of above string
tai.rport	2	Remote port number
tai.rdist	2	Remote distance indicator
tai.errno	2	Last socket error encountered
tai.herrno	2	Last name-resolution error encountered
tai.rbytes	4	Characters filled in an aborted record or line
tai.size		Size of this structure

Note that the symbol names are lower case.

Before calling TCPINF you **must** fill in the pointer and size fields for the names and addresses in the array. The TCPINF call will then fill in the referenced areas with null terminated strings.

Assuming the session obtains a connection, the remote distance indicator may contain one of the following values upon return:

Symbol	Meaning
TD$LOCALHOST	Remote is running on the local system.
TD$LOCALNET	Remote is on another system on the local network.
TD$REMOTE	Remote is on another network altogether.

Spawning Jobs

To spawn a new job with a specific configuration, use:

```
TCPSPN   error, pid, how
```

TCPSPN spawns a job using the configuration in ***how***. Upon successful creation, the new job's process ID is written to the word ***pid***. The process ID is a mechanism used by AlphaTCP to transparently keep track of jobs. The spawn descriptor referenced by how has the following structure which may be assigned using the size definition tas.siz:

Symbol	Size	Purpose
tas.mem	lword	Memory size in bytes
tas.run	lword	Pointer to command line to execute
tas.nam	lword	Pointer to desired jobname (or 0 for default)
tas.flg	word	Spawn flags. See below.

Note that the symbol names are in lowercase.

Flags which may be provided in tas.flg are:

Symbol	Purpose
TS$FNAM	Fail if name referenced by tas.nam already exists.
TS$LOFF	Start the job logged off.
TS$ADOT	Don't despawn the job when it reaches the AMOS prompt.
TS$SDOT	Suppress printing the AMOS prompt.

Terminating Spawned Jobs

Use this call to terminate a specific job and begin shutting it down:

```
TCPKIL   error,pid
```

TCPKIL removes the spawned job from the AlphaTCP process ID list and starts shutdown processing on it. The target job receives up to two SI$EXI software interrupt exit requests and Control-C's. Failing this, the job terminates with an SI$ABT software interrupt abort.

Spawned Job Cleanup

Whenever one or more spawned jobs terminate your application will receive an event. To return the status of terminated jobs, use:

```
TCPDES   error, pid, status
```

TCPDES requests status of terminated spawned jobs. Each call returns status for a single despawned job. Call TCPDES repeatedly until it indicates there is no more status to return. The exiting job's process ID is written into ***pid***, and the job's exit status (from JOBERR) is written into ***status***.

It is very important to perform this call when spawning jobs. As spawned jobs disappear, their status is held, taking up system resources. Using this call frees up the resources holding their termination status. Once you receive a child termination event, keep performing this call until you receive a TA$ENRDY error, indicating no more termination status exists. This allows for multiple jobs to have terminated for a single event.

Requesting a Session

To request control of a specific session, use:

```
TCPREQ   error, handle, newhandle
```

Requests control of the session referenced by ***handle***. Upon successful handshaking with the current owner, control will be given to the requesting job and a new handle will be written to the longword ***newhandle***.

As this is not an event-driven call, make sure that the owner job releases the session as quickly as possible to avoid any delays.

Releasing a Session

To release a specific session to the requester, use:

```
TCPREL   error, handle
```

TCPREL relinquishes control of the session referenced by ***handle*** to the requester.

Standard Filename Conversion

```
TCPCPF   error, path, fname, flags
```

TCPCPF converts the UNIX- or DOS-style path indexed by ***path*** into an AMOS filename and stores it in the buffer indexed by ***fname***. The conversion performed is documented in the *AlphaTCP User's Guide*. This is the same conversion routine used throughout AlphaTCP, providing a consistent name conversion style. When using DOS-style paths you should replace the backslashes with forward slashes first. The following options may be specified for ***flag***:

Symbol	Meaning
TF$NODIR	Discard directory information (keep filename only).

Hostname and Address Conversion

Normally a hostname is translated to an address automatically when establishing a connection. Likewise, an address is translated to a hostname prior to notification of a new incoming connection.

If you find a reason you need to perform this conversion outside of an actual connection you may use:

```
TCPRES   error, handle, string, flags
```

This will start a domain name query on ***string***. When the query is complete, a TE$RESOLVED event will be generated. The resolved name or address may be read using TCPINF. You should close the handle when you are done with the query.

flags controls how the query is performed. The following options may be specified for ***flag***:

Symbol	Meaning
TR$REV	Used when string contains an IP address and you want to find the hostname

EVENTS

The following section contains details for each of the events generated.

Global Events

Symbol	Meaning
TG$CHILD	One or more child processes have terminated. Use the TCPDES call to find out which and the exit status.
TG$PARENT	The parent process has terminated. This may or may not indicate an error, depending upon the application.
TG$ATTENTION	Another process is requesting attention. Actual communication between processes is left up to the application. This is simply a signaling mechanism with a built-in ability to pass a pair of longword values. Use TCPQAT to determine the signaling process and any passed arguments.
TG$DOWN	The TAME subsystem is shutting down. This is a fatal error, all session handles are now invalid. Any existing connections will be reset. Most often you won't see this event as the subsystem will usually be gone by the time you perform the TCPEVT call. In this case your TCPEVT will return TA$ENAVL (TAME unavailable) status instead.

Session Events

Symbol	Meaning
TE$CONNECTED	An outgoing connection request has completed successfully, or an incoming connection has been established.
TE$DISCONNECTED	The remote application has closed the connection. This results from an orderly shutdown.
TE$DROPPED	A pending connection request has failed, or an active connection has been terminated. This usually results from a fatal network error, although the remote application itself can request that the connection be aborted in some cases.
TE$DATAIN	There is data available on the connection ready to be read. You should only perform a single read per TE$DATAIN event. If you are reading records using TCPFIL or lines using TCPLIN you should ignore TE$DATAIN events, possibly using them to restart record or line mode instead.
TE$DATAOUT	The connection is ready to have data written to it and any previously sent buffer may now be reused. You should only perform a single write per TE$DATAOUT event. Also, the buffer used to send data should remain unaltered until the next TE$DATAOUT event.
TE$RECORD	A complete record or line is now available. This implies either a TCPFIL or TCPLIN operation was being performed on the connection. The connection reverts back to normal mode and will begin generating TE$DATAIN events again. To begin reading another record, initiate another TCPFIL operation. To begin reading another line, initiate another TCPLIN operation. A convenient place to do this is during TE$DATAIN processing.
TE$HANDOFF	A child job is requesting control of an existing session. Perform a TCPREL to relinquish the session to the child.
TE$TRUNCATED	Connection closed with a partially filled line or record
TE$RESOLVED	Name or address query has completed

QUICK REFERENCE LIST OF CALLS

The following alphabetic list includes the format and a brief description for all uses of TAME:

Calling Format		Description
TCPACC	ERR,OHAND,NHAND	Accept a connection and generate a new handle.
TCPCNI	ERR,HAND,HOST,DPORT,SPORT,FLAGS	Establish a connection using immediate strings.
TCPCON	ERR,HAND,HOST,DPORT,SPORT,FLAGS	Establish a connection using indexed strings.
TCPCPF	ERR,PATH,FNAME,FLAGS	Convert a path to an AMOS filename.
TCPDSC	ERR,HAND,RST	Disconnect the session.
TCPDES	ERR,PID,STAT	Handle despawned job cleanup.
TCPEDN	ERR,HAND	Notify completion of current event driven activity.
TCPEVT	ERR,HAND,GEVENTS,SEVENTS	Request current events for session.
TCPFIL	ERR,HAND,BUF,RECLEN	Begin filling a record of reclen bytes.
TCPINF	ERR,HAND,INFO	Return information about current connection into an array and strings.
TCPKIL	ERR,PID	Kill a spawned job.
TCPLIN	ERR,HAND,BUF,MAXLEN	Begin reading a line of up to maxlen bytes.
TCPLNI	ERR,HAND,PORT,FLAGS	Listen for connections using immediate strings.
TCPLSN	ERR,HAND,PORT,FLAGS	Listen for connections using indexed strings.
TCPPID	ERR,PID,PPID	Request process ID information.
TCPPOL	ERR,GEVENTS,ARRAY,MAX,FILLED	Request current events for all sessions.
TCPQAT	ERR,PID,ARG1,ARG2	Request attention event information.
TCPRED	ERR,HAND,BUF,MAXLEN,LEN	Read currently available data up to maxlen bytes.
TCPRES	ERR,HAND,STRING,FLAGS	Resolve a hostname or an address
TCPREL	ERR,HAND	Release control of a session to a requester.
TCPREQ	ERR,HAND,NHAND	Request control of a session.
TCPSAT	ERR,PID,ARG1,ARG2	Set an attention event for the given process.
TCPSPN	ERR,PID,HOW	Spawn a job.
TCPWRT	ERR,HAND,BUF,LEN	Write data for len bytes.
TCPWAT		Wait for something to do.

TAME ERROR CODES

There are three sets of error codes related to TAME:

- The ***general*** error codes may be returned in the ERR argument in any TCP call. The TASYM.M68 file defines symbols for the general error codes so you can test for them easily in your application.
- The ***socket*** error codes contain information about the BSD socket function, and can be accessed using the TCPINF call.
- The ***name*** error codes are also accessed using TCPINF. They contain information about the ***DNS Resolver*** functions.

General Error Codes

Symbol	Meaning
TA$ENAVL	TCP interface not available
TA$ENVLD	Invalid TCP function call
TA$EINTR	TCP function interrupted
TA$ENRDY	Not ready for requested TCP operation
TA$EDISA	TCP session has been disabled
TA$ESOOR	TCP service out of resources
TA$ESOOM	TCP service out of memory
TA$ESOCK	General TCP socket failure
TA$EPORT	Invalid TCP service or port
TA$EHOST	Invalid TCP host name
TA$EHUNR	TCP Host unreachable or not responding
TA$EPARM	Parameter error in TCP call
TA$ETDV	Spawn failure, cannot find tdv
TA$EIDV	Spawn failure, cannot find idv
TA$EMEM	Spawn failure, not enough memory
TA$EFUL	Spawn failure, job table is full
TA$EDUP	Spawn failure, job already exists
TA$ELIC	Spawn failure, would exceed AMOS license

Socket Errors

Decimal Value	Meaning	Decimal Value	Meaning
1	File specification error	523	Address family not supported by
2	Insufficient free memory	524	Socket type not supported
3	File not found	525	Protocol not supported
4	File already exists	526	No buffer space available
6	Device full	528	Socket is already connected
7	Device error	529	Socket is not connected
8	Device in use	530	Bad protocol option
10	Protection violation	531	Connection reset by peer
11	Write protected	532	Software caused connection abort
12	File type mismatch	533	Network is down
13	Device does not exist	534	Connection refused
14	Illegal block number	535	Host is unreachable
27	Remote is not responding	536	Protocol wrong type for socket
28	File in use	537	Operation not supported on socket
30	Deadly embrace possible	538	IP subnet table full
500	Device not a stream	539	Subnet module not linked
501	No data	540	Unknown ioctl call
502	Timer expired	541	Failure in streams buffer allocn
503	Out of streams resources	542	ICMP protocol unreachable
504	Machine is not on the network	543	ICMP port unreachable
505	Package not installed	544	ICMP network unreachable
506	The object is remote	545	Protocol family not supported
507	The link has been severed	546	Can't send after socket shutdown
508	Advertise error	547	Network dropped connection on reset
509	Surmount error	548	Destination address required
510	Communication error on send	549	Invalid Ethernet packet
511	Protocol error	550	Type registration error
512	Multihop attempted	551	Illegal address
513	Inode is remote	552	Message too big for buffer
514	Cross mount point	553	No message of requested type exists
515	Trying to read unreadable message	554	File table overflow
516	Given log. name not unique	1000	USAM error

Decimal Value	Meaning	Decimal Value	Meaning
517	f.d. invalid for this operation	1001	Domain error
518	Remote address changed	1002	Range error
519	Try again	1003	Interrupted system call
520	Socket operation on non-socket	1004	Too many open files
521	Can't assign requested address	1005	No such device or address
522	Address already in use	1006	Not super user

Name (DNS Resolver) Errors

Value	Meaning
1	Unknown host
2	Host name lookup failure
3	Unknown server error
4	No address associated with name

EXAMPLES

A number of example programs are provided in the [7,7] account of the AlphaTCP release. Please refer to the TCP.DIR file of the release.

Appendix A
Disk Structure Format

AMOS supports a flexible disk file system that relieves you of the task of keeping track of files, links, and block counts. Not only does this hiding of details make it easier to create programs which access files, but it also allows a program to deal with different file systems without regard for the low-level details.

AMOS supports two distinct file systems: the *traditional* file system, which uses 16-bit pointers and limits file sizes to 32 MB or less; and the *extended* file system, which uses 32-bit pointers, allowing file sizes up to 2,097,512 MB. Both file systems can be present on a given system at the same time, with different logical units containing different file systems. You can choose how to set up each logical unit based on the requirements of a given system.

The structure of the disk directory used by AMOS is described here for informational purposes only; all manipulation—reading or writing—of the directory structure should be done via the standard AMOS file and directory handling calls. By using only these calls to access directories, you can ensure that your program will not be affected by future changes to the underlying file structures.

PHYSICAL BLOCK FORMAT

The logical block size for all disks used within the AMOS file structure, regardless of type, is 512 bytes. While most disk drives are formatted by AMOS to have a physical sector size of 512 bytes, AMOS provides support for devices with smaller physical sector sizes by automatically clustering the smaller sectors into a 512 byte block.

Regardless of the physical sector size, all AMOS file system I/O is done with 512 byte disk blocks.

DISK BLOCK TYPES

The AMOS file systems use five different block types which are categorized by their use in the logical processing of files. Each block is 512 bytes long, but their internal structure differs due to different usage in the system. The five block types are:

- Disk Label block
- Bitmap blocks
- Directory blocks
- Sequential file data blocks
- Contiguous file data blocks

Physical blocks 0, 1, and 2-n (where n depends on the size of the disk) contain predefined information, regardless of file system type. Block 0 contains the disk label block, block 1 contains the first disk directory block, and blocks 2-n contain the disk storage allocation bitmap.

The Disk Label Block

The Disk Label block is always block 0 and is used by the LABEL and MOUNT programs, plus any other programs that need to know what disk is currently mounted on a given disk drive. The label block is reserved for disk identification information. It is permanently allocated, so a system routine will not accidentally use it as a data block. Since this block is reserved for the disk label, you should not attempt to use it for other purposes.

The label block is used to store the flags that indicate to AMOS what directory type (traditional or extended) is in use on a logical unit. Other than the presence (or absence) of these flags, the label block format is the same for both file systems.

The format of the disk label block is as follows:

Symbol	Meaning	Size	Notes
LB.HDR	Header	2 words	^O125252 : 052525
LB.VLN	Volume Name	40 bytes	ASCII text
LB.VID	Volume ID	10 bytes	ASCII text
LB.CRE	Creator	30 bytes	ASCII text
LB.INS	Installation	30 bytes	ASCII text
LB.SYS	System Name	30 bytes	ASCII text
LB.CRD	Creation Date	4 bytes	Separated date
LB.ACD	Access Date	4 bytes	Separated date
	Unused, reserved	36 bytes	
LB.FLG	Flags	1 byte	Bitmask

Symbol	Value	Meaning
LB$14D	4	Disk contains extended format directories

	Unused, reserved	5 bytes	

The LB.HDR field contains two "noise" words used to identify the disk as being labeled. If these two words do not contain the values specified above, the disk is not considered to be labeled.

All ASCII text fields are terminated by either a null or by the field length.

The dates contained in the disk label in system separated format, as shown in Figure A-1. Separated date format is described in more detail in Chapter 10.

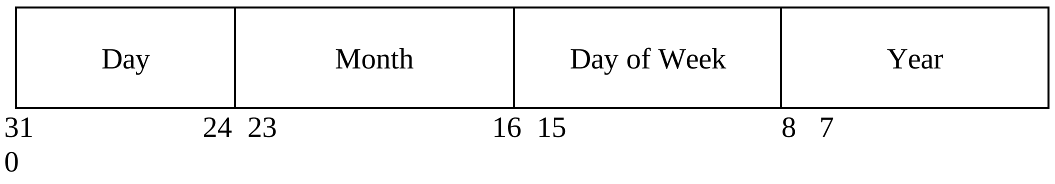


Figure A-1: Disk Label Date Format

The Bitmap

The bitmap consists of one or more disk blocks that are used as a storage allocation map for the disk. The bitmap always starts at block two and extends for as many blocks as are necessary to map the entire disk. Each word in the bitmap is capable of representing the state of 16 logical blocks with one bit being used for each block. The bit is set if the block is in use and cleared if it is free. The last two words of every bitmap are a double-word hash total that is used to maintain bitmap integrity during processing. Any remaining words in the last bitmap block are unused. The bitmap itself is permanently allocated but contains no links to other system disk blocks. If you destroy the bitmap, you can run the DSKANA program to recover it.

The format of the bitmap is the same for both traditional and extended format directories.

Directory Blocks

Directory blocks are used by both file systems to define both the data files stored on the disk and the directory or account (PPN) structure in which these data files are organized.

While the implementation details are different, both file systems use a multi-level organization allowing collections of files to be grouped into accounts (PPNs). Both start their directory structure with block 1 of the disk.

Traditional Format Directories

The traditional format directory structure makes use of a two level directory hierarchy consisting of a master file directory (MFD) which in turn points to user file directories (UFD) allocated throughout the disk structure.

The master file directory block always starts in block 1 and forms the basis of the file structure organization. It contains one entry of four words for each user PPN which is allocated to this disk by the SYSACT program. If more entries are needed, additional disk blocks may be linked to block 1 to extend the size of the MFD.

User directory blocks (UFDs) contain up to 42 entries of six words each to describe user files in a PPN. The first word of each directory block is a link word to the next directory block, to accommodate situations where more than 42 files are allocated in the current user area. The final directory block has a zero link word, indicating that no more directory blocks follow.

Extended Format Directory Blocks

Extended format directory blocks consist of a multi-level hierarchy of directories. The root of these directories always starts in block 1 of the disk. This root directory contains a series of variable length directory entries defining additional directories (PPNs) on the disk. In turn, each of these directories contains variable length entries defining the individual data files on the disk.

While the structure of the extended format directory blocks does not require it, for compatibility reasons the file system is formatted as a two-level structure much the same as the traditional format file system.

Sequential File Data Blocks

Sequential data files consist of a series of disk blocks linked together by a disk block pointer contained at the start of each block. Traditional format sequential files have a two byte link and 510 data bytes in each block. Extended format sequential files have a four byte link and 508 data bytes in each block. The link contains the block number of the next block in the file. A zero link indicates this is the last block in the file. The last block in the file may have anywhere from 0 to 510 (0 to 508 in extended format) active data bytes in its data area. The directory block item contains this number. Sequential files are normally processed as one long string of bytes from start to finish.

Contiguous File Data Blocks

Contiguous file data blocks have 512 data bytes and no links. Contiguous files must be allocated as a group of blocks with no intervening blocks belonging to other files. They must be allocated before you use them, while sequential files are allocated one block at a time as required. Contiguous files allow random access processing, since any block may be located as a direct offset relative to the base block.

Contiguous files are structured the same under both the traditional and extended file systems.

TRADITIONAL FORMAT FILE STRUCTURE

The traditional format file structure is depicted in Figure A-2 and resembles a waterfall with the MFD as its source. The MFD block has one entry for each allocated user on this disk. The MFD contains as many blocks as are necessary to hold all the PPNs on the disk. Each MFD item then contains the block number of the first user directory block for that PPN number. The user directory block has one item for each data file in this user's area. Each directory item then contains the block number of the first data block in the file. Sequential files then chain through the data blocks by link words as shown in the diagram. The two files that are partially depicted are LOG.LIT and DIR.LIT in user area [1,4], which happens to be the system program area. Contiguous files have no link words and must occupy physically adjacent blocks beginning with the first block as addressed in the directory item.

MFD Item Format

The MFD consists of one or more linked blocks, each of which contains space for 63 entries. The first MFD block is always block 1. Additional blocks are linked to this block as new PPNs are added.

Each of the 63 items in an MFD block is four words long and contains the PPN specification, user directory link, and password. The format of the item is:

Word Offset	Meaning
1	User PPN (project and programmer are each one byte)
2	Block number of first user directory block
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 blocks 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 blocks are linked together by means of an 8-byte entry at the end of each MFD block. The fields in this entry are:

Symbol	Size	Meaning
MF.ZER	2 bytes	Always zero
MF.NXT	2 bytes	Link to next block of MFD
MF.PRV	2 bytes	Link to previous block of MFD
MF.FLG	2 bytes	Flags:
		MF$PRM = This block is permanent

MF.NXT is zero in the last block of the MFD. MF.PRV is zero for single block MFDs; otherwise, the first block contains the number of the last block in the MFD.

MFD items are added, deleted, and changed through use of the SYSACT program.

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:

Word Offset	Meaning
1-3	Filename.extension of the file packed RAD50
4	Number of data blocks in this file
5	Number of active data bytes in last block
6	Block number of first data block 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 also represents the negative active byte count of the file if the contiguous file has been opened for output and has been written into sequentially.

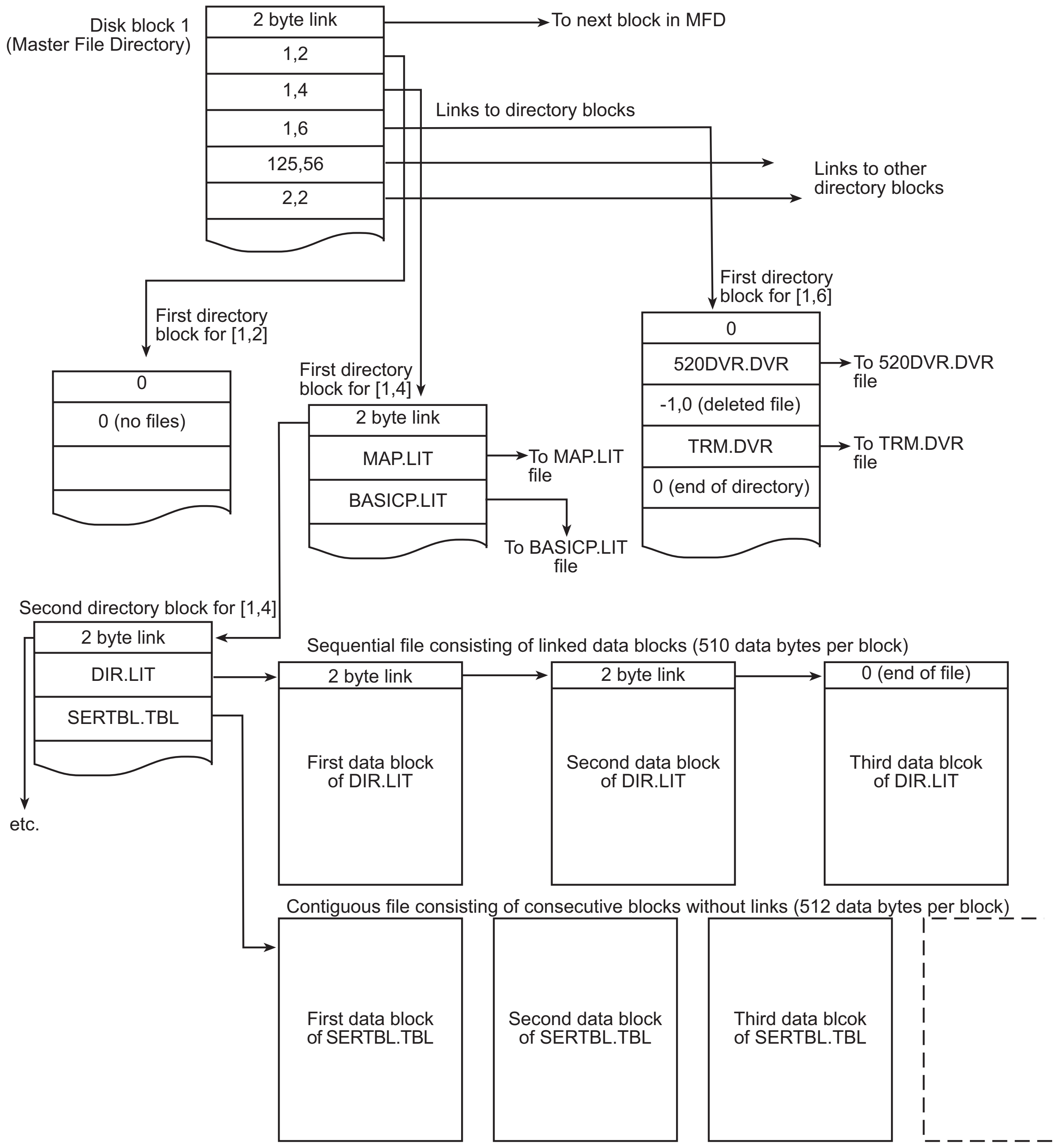

Disk block 1
(Master File Directory)
2 byte link
To next block in MFD
1,2
1,4
Links to directory blocks
1,6
125,56
2,2
Links to other directory blocks
First directory block for [1,6]
First directory block for [1,2]
0
0 (no files)
First directory block for [1,4]
2 byte link
MAP.LIT
To MAP.LIT file
BASICP.LIT
To BASICP.LIT file
0
520DVR.DVR
To 520DVR.DVR file
-1,0 (deleted file)
TRM.DVR
To TRM.DVR file
0 (end of directory)
Second directory block for [1,4]
2 byte link
DIR.LIT
SERTBL.TBL
etc.
Sequential file consisting of linked data blocks (510 data bytes per block)
2 byte link
First data block of DIR.LIT
2 byte link
Second data block of DIR.LIT
0 (end of file)
Third data blcok of DIR.LIT
Contiguous file consisting of consecutive blocks without links (512 data bytes per block)
First data block of SERTBL.TBL
Second data block of SERTBL.TBL
Third data blcok of SERTBL.TBL

Figure A-2
Traditional Format Disk File Structure

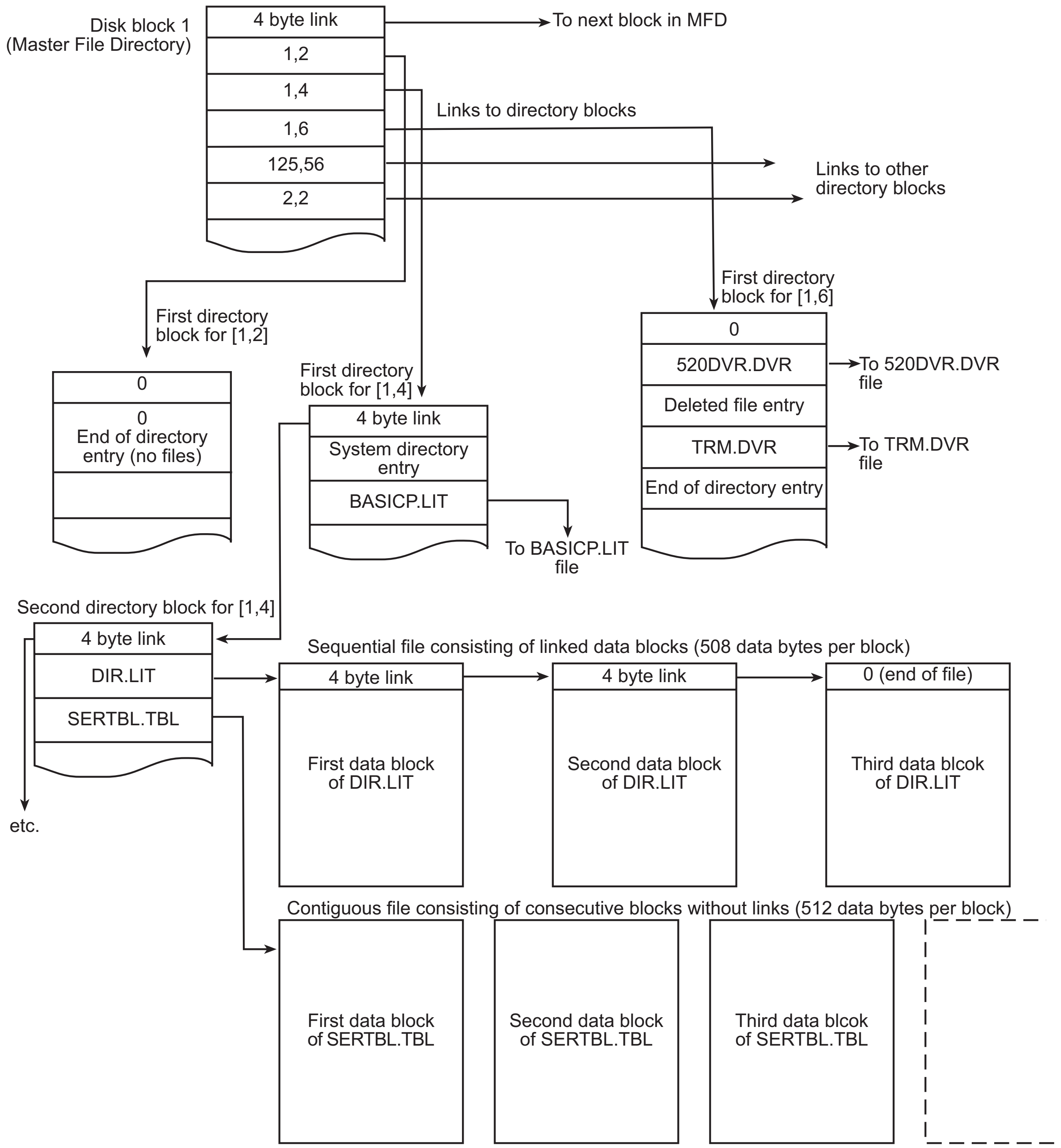

Disk block 1
(Master File Directory)
4 byte link
To next block in MFD
1,2
1,4
1,6
125,56
2,2
Links to directory blocks
Links to other directory blocks
First directory block for [1,2]
0
0 End of directory entry (no files)
First directory block for [1,4]
4 byte link
System directory entry
BASICP.LIT
To BASICP.LIT file
First directory block for [1,6]
0
520DVR.DVR
To 520DVR.DVR file
Deleted file entry
TRM.DVR
To TRM.DVR file
End of directory entry
Second directory block for [1,4]
4 byte link
DIR.LIT
SERTBL.TBL
etc.
Sequential file consisting of linked data blocks (508 data bytes per block)
4 byte link
First data block of DIR.LIT
4 byte link
Second data block of DIR.LIT
0 (end of file)
Third data blcok of DIR.LIT
Contiguous file consisting of consecutive blocks without links (512 data bytes per block)
First data block of SERTBL.TBL
Second data block of SERTBL.TBL
Third data blcok of SERTBL.TBL

Figure A-3
Extended Format Disk File Structure

EXTENDED FORMAT FILE STRUCTURE

The extended format directory structure consists of a series of linked directory blocks, each containing variable length entries. Each of these entries defines either another level of directories, or a data file. Figure A-3 illustrates how these different directory blocks link together.

While the extended format directory structure allows directories and data files to be freely mixed within a given directory block, most AMOS software is limited to dealing with the two level directory structure originally imposed by the traditional format structure. In this mode, disk block 1 (and the blocks it links to) contains only directory items, each one defining a PPN. Each of these directory items points to a series of directory blocks (a UFD) containing only data file items.

Directory Block Format

All extended format directory blocks share the same format. Each consists of a four-byte link (at the base of the block) followed by 508 bytes of directory entries. Each directory entry is of a variable length, producing directory blocks with differing numbers of entries in each block.

Each directory entry is one of four types: a *system entry* used to define internal file system related information and linkages; a *directory entry* defining a link to a lower level of directories; a *data file entry* defining a link to a data file; or a *deleted entry* reserving available space previously occupied by some other entry type. System and directory entries share a common format, while data file entries differ in part of their format.

Figure A-4 illustrates the format of a system or directory entry, while Figure A-5 illustrates the format of a data file entry. The contents of each of these fields is defined below:

Symbol	Meaning
D$TYP	This 32-bit field is broken into three sub-fields. The high-order word of D$TYP contains flags describing this entry. The low-order byte of D$TYP contains the size of the name field for this entry, while the high-order byte of the low-order word contains flags specifically pertaining to the filename format of this entry.
D$PRT	This 32-bit field contains the protection level associated with this entry.
D$DAT	This field consists of three 32-bit packed dates representing the creation, update and backup dates for this entry. These dates can be manipulated by the $PAKDT and $UNPDT library routines.
D$NXT	This 32-bit field, used only in system and directory entries, contains a link to the next lower directory level.
D$CUR	This 32-bit field, used only in system and directory entries, is reserved for future use.
D$PRV	This 32-bit field, used only in system and directory entries, is reserved for future use.
D$BAS	This 32-bit field, used only in data file entries, contains the block number of the first block in the data file.
D$FSZ	This 32-bit field, used only in data file entries, contains the number of disk blocks occupied by the data file.
D$RSZ	This 16-bit field, used only in data file entries, contains the default record size used by this file. This field is only valid for contiguous files.
D$LSZ	This 16-bit field, used only in data file entries, contains the number of active bytes contained in the last block of a sequential file. This field will contain a -1 for contiguous files.
D$NAM	This variable length field contains the directory name (PPN) in the case of directory entries, and the filename in the case of data file entries. The size of this field (in bytes) is defined in the D$TYP field. For standard six-byte directory entries, the first word will contain the PPN, with the next two words containing the password, packed RAD50. For standard six-byte data file entries, the filename and extension will be packed RAD50 in this field.

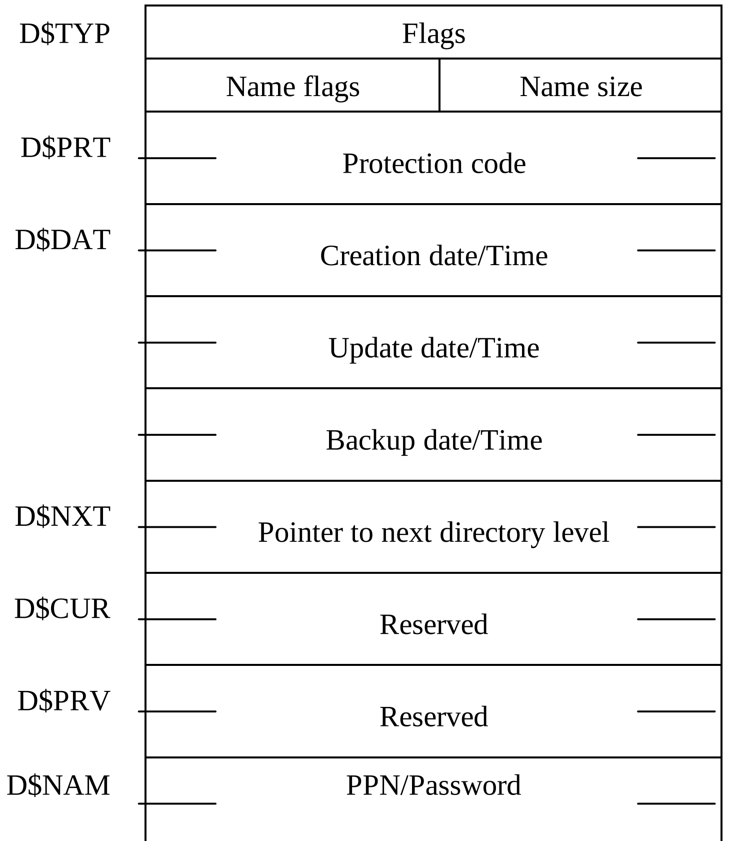


Figure A-4: Extended Format System and Directory Entry Format

The flags in the high-order word of D$TYP are defined as follows:

Octal Value	Hex Value	Meaning
1	1	This item is a system entry
2	2	This item is a directory entry
4	4	This item is a data file entry
20	10	This data file entry is an initialized USAM file
100000	8000	This item is a deleted entry

The size of the name field within D$TYP defines the number of bytes contained in this entry past the D$NAM field. For directory items this is typically six, as it is for data file items.

The only flag defined in the filename flag field is bit 0 which indicates that the filename consists of an ASCII string, rather than a packed RAD50 value. This flag is only valid for data file entries.

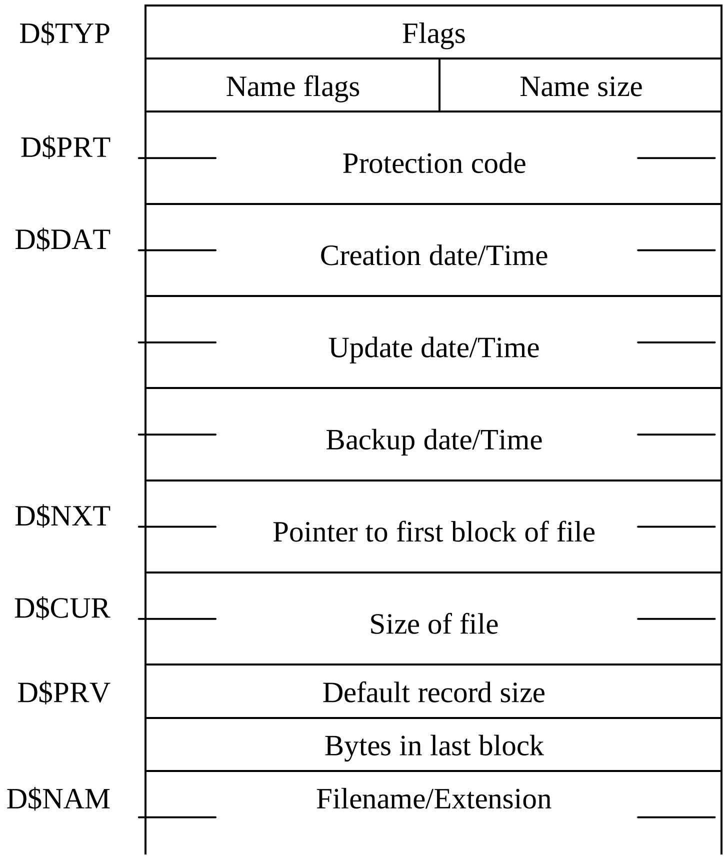


Figure A-5: Extended Format Data File Entry Format

PROGRAM HEADER FORMAT

AMOS supports the use of optional file headers for program (.LIT) files. This option gives you the ability to attach version numbers, assign required privileges, and identify characteristics of program files.

A program file that contains a header is flagged by one of two control values in the first word of the file. If the first word is not one of the two special values, then the operating system assumes that no program header exists.

The format of the program header (as defined in SYSSYM.UNV) is:

Symbol	Octal Byte Offset	Hex Byte Offset	Size	Meaning
PH.FLG	0	0	2 bytes	Flag: **Value** / **Meaning** -1 Program must be run logged in. -2 Program can be run logged out
PH.VER	2	2	4 bytes	Packed version number
PH.PRV	6	6	2 bytes	Required privilege bits: **Symbol** / **Meaning** PV$RSM Can read system memory PV$WSM Can write system memory PV$RPD Can read physical disk PV$WPD Can write physical disk PV$DIA Can run diagnostics PV$PRV Can change privilege bits
PH.CHR	10	8	2 bytes	Program characteristics: **Symbol** / **Meaning** PH$REE Program is re-entrant PH$REU Program is reusable PH$OPR Program must be logged under OPR: (DSK0:[1,2]) PH$L12 Program must be logged into [1,2] PH$M20 Program requires a MC 68020 or later processor PH$ILC Program does not want the command line or other terminal input folded into upper case PH$EXT Program wants extended (8-bit) character input PH$ERR Program inspects JOBERR field: do not clear JOBERR on initial program load

The size of the header area is defined as PH.SIZ. The privilege bits should be defined in each program but are reserved for future use. The program characteristics are checked by the LOAD and SYSTEM programs to output warning messages if an attempt is made to LOAD a non-re-usable program or SYSTEM a non-re-entrant program.

The version number is stored in a packed 32-bit format. The display format of the version number is:

```
A.BC(D)-E
```

Component	Meaning	Symbol	Range (Decimal)
A	Major version number	VMAJOR	0 - 255
B	Minor version number	VMINOR	0 - 15
C	Sub-version letter	VSUB	0 – 15 (null – N)
D	Edit number	VEDIT	0 - 4095
E	Patch level	VWHO	0 - 15

The major version number reflects a major revision to the program, including operational changes. The minor version number flags a change that may include additional features, but does not change the overall operation of the program. The sub-version marks each formal release of a program, where changes may be very minor. The edit number is incremented each time the program is modified, no matter how minor the change, and continues to increment forever, never being reset. The patch level records the number of patches installed in this version of the program.

Defining the Program Header

To define a program header area, use the PHDR macro. It ***must*** be the first instruction in the program. The format of the macro is:

```
PHDR    flag, privilege, characteristics
```

flag must be -1 or -2, ***privilege*** is the required privilege bits, and ***characteristics*** are the characteristics bits defined above. Before calling the PHDR macro, you should set up the version number by defining values for VMAJOR, VMINOR, VSUB, VEDIT, and VWHO. Any of these that are not defined will default to zero values.

For example, to define the program header for a program whose version number is "1.2B(101)" and which requires the ability to read physical disk blocks, and is both re-entrant and re-usable, and requires the user to be logged in, would appear in source code as:

```
        VMAJOR=1.
        VMINOR=2.
        VSUB=2.
        VEDIT=101.

START:  PHDR     -1,PV$RPD,PH$REE!PH$REU
        code
```

You can use the VCVT monitor call to unpack and display version numbers. See section 8.3.5 for more information on the VCVT call.

Appendix B
Terminal Service System

The terminal service system incorporated within AMOS is a flexible and efficient set of routines and drivers for interfacing a variety of different terminals with different interface boards. Not only does it support the full range of Alpha Micro supplied interfaces, but you may also write your own drivers for terminals and interfaces not supported by Alpha Micro. This appendix describes the general structure and function of the terminal service system and some of the data structures used, but does not go into details on how to write user-defined drivers. Source code for several terminal and interface drivers are provided for those individuals who want to write their own driver programs. For more specific information on the terminal service system and how terminal drivers may be written, see the *AMOS Terminal Service System User's Guide*. For a general overview of the Alpha Micro Operating System, see Part III of *Introduction to AMOS*.

GENERAL STRUCTURE

AMOS contains a general terminal processing routine called TRMSER whose function is to link user programs and monitor processes to the outside world of interactive terminals; this is done purely on a data basis, without regard to terminal or interface hardware. TRMSER processes data on a character-by-character basis. Monitor calls are available to your programs for passing characters and full buffers of data between the terminals and the system. Think of TRMSER as a telephone operator who switches calls back and forth between sources and destinations without regard to the type of telephone in use or the name of the person using that telephone. TRMSER also provides the synchronous link to the asynchronous world of the terminal hardware.

TRMSER is a monitor routine that is embedded in the operating system skeleton monitor. In addition to the general TRMSER routine in the monitor, there must exist one or more routines called drivers that take the data from TRMSER and translate it as necessary into the specific codes required by the hardware, and then route it to the terminal through the interface board. These drivers reside in account [1,6] of the system disk, and are automatically loaded into system memory in response to the terminal definition (TRMDEF) command lines in the system initialization command file at the time of system bootup. Driver programs MUST be re-entrant; only one copy of a driver is loaded into memory regardless of the number of terminals or interface boards of that type defined on the system.

The terminal service system uses two general types of drivers: interface drivers and terminal drivers. Interface drivers contain the routines necessary to get data characters to and from the physical interface hardware. Terminal drivers contain routines that process each character that goes to or from the terminal. Terminal drivers handle code character conversions, echoing functions, line-feed null characters, cursor control, and special functions as required by the type of terminal in use.

INTERFACE DRIVERS

Interface drivers link the TRMSER routines and the actual hardware responsible for getting characters to and from the terminal device. The interface drivers are assembly language programs with an .IDV extension. The filename of the interface driver appears in the TRMDEF command line of the system initialization command file, and tells the system what kind of interface is being used by the terminal defined by that command line. The interface driver handles all initialization sequences for the ports, and also sets up the interrupt handling used by the ports. A special interface driver exists on the system called the PSEUDO interface driver; it controls no hardware at all, but instead represents a software-controllable interface for inter-job communication and control.

Interface Driver Format

The following is a typical AMOS .IDV communications area. The program header is stripped off when it is loaded into memory by TRMDEF.

```
IDVBAS: PHDR     -1,0,0        ; Program header at base of IDV
        BR       CHROUT        ; character output routine
        BR       10$           ; initialization routine
        BR       GETPIN        ; read RS-232 input pin status routine
        BR       SETPIN        ; toggle RS-232 output pin routine
        WORD     ^H0A5A5       ; noise word to signal flags word next
        LWORD    FLAG          ; IDV characteristics
        BR       20$           ; disable/enable receiver routine

10$:    JMP      INIT          ; intermediate jump to init routine
```

Offsets to each element of the communications area are defined by the following symbols.

Symbol	Octal Byte Offset	Hex Byte Offset	Purpose
ID.CHR	0	0	First instruction of character output initiation routine, usually a branch instruction.
ID.INI	2	2	First instruction of port initialization routine, usually a branch instruction.
ID.RST	4	4	First instruction of RS-232 input pin status routine, usually a branch instruction.
ID.WST	6	6	First instruction of RS-232 output pin toggle routine, usually a branch instruction.
ID.NOI	10	8	16-bit noise word
ID.FLG	12	A	32-bit IDV characteristics flag
ID.SPC	16	E	First instruction of special function routines, usually a branch

The IDV characteristics flag bits are:

Symbol	Meaning
ID$ASNC	Asynchronous type interface
ID$SYNC	Synchronous type interface
ID$TRTS	Transmitter can control the RTS output
ID$RRTS	Receiver can control the RTS output
ID$CTS	The CTS input can enable transmitter
ID$BGEN	Interface can generate breaks
ID$DRCV	Can disable receiver interrupts
ID$SMRT	IDV has intelligent capabilities (e.g., AM350)

TERMINAL DRIVERS

Terminal drivers customize the handling of character input and output based on the type of terminal being used. They are assembly language programs that have the .TDV extension. The filename of the driver is the name by which the terminal type is referenced in the TRMDEF statements in the system initialization command file. Typical terminal drivers are for the AM-65 and AM-75.

The terminal driver processes all input and output characters, and determines if these characters need special handling because of the type of terminal being used. The terminal driver handles echo control and different methods of character deletion. For example, most CRT terminals have the ability to back up and erase the character being deleted, while hard copy terminals must explicitly echo the character, usually in a format that distinguishes the characters from those accepted as input. The terminal driver is responsible for using the current form of deletion echo.

Terminal drivers may also be written for software-controlled ports, and two such drivers are built into the monitor already. The PSEUDO and NULL terminal drivers are used in conjunction with the PSEUDO interface driver, and provide a means for passing characters straight through to the controlling job or discarding output characters that are unimportant. Terminal drivers are usually unconcerned with the type of interface used to physically tie the terminal to the computer.

INTERSYSTEM DRIVER LINKS

The relationship between the different elements of the terminal service system can seem confusing at first; nevertheless, efficient systems-level programming requires a thorough understanding of the links that exist between these items. The following units are referenced in further discussions:

- JOB - A job is the unit that controls the operation of one task or a series of tasks running on the system. A job is independent of any other jobs running on the system unless it is tied to them by special user software. Every job on the system has a unique name one to six characters long.
- TERMINAL - A terminal is the hardware device used to physically transfer data into the system, and get data from the system to the user on a character-by-character basis. Terminals do not themselves have names. Typical terminals might be an AM-65 or AM-75.
- TERMINAL CONTROL BLOCK - A terminal control block (TCB) is a block of memory in the system area set up by a TRMDEF statement. It is the basic unit by which a terminal in the system is referenced when attaching that terminal to a specific job, or when using the terminal as an I/O device under control of the TRM device driver. The terminal control block has a unique name one to six characters long.

- INTERFACE DRIVER - An interface driver is the program that transfers characters back and forth between the terminal and the hardware interface board to which the terminal is physically connected. The interface driver has a name one to six characters long that is referenced by the TRMDEF statements in the system initialization command file. Interface drivers reside in account DSK0:[1,6], and have the extension .IDV.
- TERMINAL DRIVER - A terminal driver is the program that performs the character code conversions required by the terminal in use. The program has a name (one to six characters long) that is referenced only in the TRMDEF statement of the system initialization command file. Terminal drivers reside in account DSK0:[1,6] and have an extension of .TDV.
- DEVICE DRIVER - A device driver is a program that allows the system to communicate with any I/O device connected to the system. Device drivers are written for disks, tape units, printers and terminals. The handling of terminals as devices for use by the generalized file service system is done through the TRM device driver, and not through the terminal drivers themselves. Device drivers have a one to three character name that is referenced in the device table statement (DEVTBL) in the system initialization command file, and in user file specifications (e.g., AMS1:FILNAM.TXT). Device drivers reside in account DSK0:[1,6] and have the extension of .DVR.

The terminal control block contains the links to the defined interface driver and to the defined terminal driver; it thus is the basic unit by which terminals are referenced on the system. When a terminal is attached to a job, the JCB (Job Control Block) and the TCB become linked to each other. A job is considered to be detached if it is not linked to a TCB, and a terminal is considered to be detached if it is not linked to a JCB. A job may only be linked to one controlling terminal, and vice versa.

A job performs I/O operations through the particular device driver referenced by the device specified in the file specification. A job performs terminal operations through the linked TCB for the terminal that is controlling that job. A detached job is placed into terminal wait state if it attempts to perform a terminal input or output operation. Since I/O operations differ in structure and usage from terminal operations, performing I/O operations to a terminal must be done through some mechanism other than directly into the TCB. From a system standpoint, the TCB works differently than a device driver. To allow generalized I/O, without regard to whether it is to a terminal or other device, a general device driver has been written called TRM which allows terminals to be accessed as devices, as opposed to being accessed only as job controlling terminals. This operation will be described later.

Terminal Input Characters

Terminal input characters are processed through a complex chain of events. When a terminal keyboard character is struck by the operator, it is transferred to the hardware interface which then passes it to the interface driver routine. The interface driver routine reads in the character and then passes it to the TRMSER processor. TRMSER puts the character into the input buffer to wait for pickup by the program or monitor. As an asynchronous event, if echoing is not suppressed or is local to the terminal, TRMSER passes the character back to the terminal driver (when it is about to be echoed) to again allow the terminal routine to perform special functions. An example of this is the special echoing of Control-U characters for line deletion or rubouts for character deletion. The terminal routine then passes the character (or the converted character) back to TRMSER to be sent to the output processor.

Terminal Output Characters

Terminal output characters can come from two main sources: 1. characters to be echoed from the input processor; and, 2. characters to be output (generated by the monitor or user program) as messages or data to the user. Both are handled differently from a buffering standpoint, but eventually are presented to a common output routine in TRMSER to be sent to the terminal. Each character for output goes from TRMSER to the terminal driver for possible output code conversion or character translation. An example of this would be the null sequence sent after every line-feed for timing purposes to some printing terminals that require a delay to allow for mechanical settling. The terminal driver processes the character and then sends it back to TRMSER for position processing. TRMSER then passes the output character (or converted character) from the terminal driver to the interface driver where it is physically output to the terminal.

USING TERMINALS AS I/O DEVICES

Most programs (including the print spooler) perform input and output operations to I/O devices rather than to the controlling terminal. In some instances it is desirable to perform these operations on a terminal rather than a specific I/O device defined by its own device driver. One example would be the printing of data on a printing terminal, or the use of these terminals as the output device of the printer spooler. Any terminal may be accessed as a device through the general device driver called TRM. The TRM device driver acts as a software link between the format required by the FILSER file service system and the TRMSER terminal service system. Any terminal can be considered a device by using the device code TRM and using the name given the terminal definition unit as the filename (the extension and PPN are ignored in the file specification).

For example, suppose you have an Epson printer connected to an AM-355 using port number two. Your TRMDEF command in the system initialization command file might look like this:

```
TRMDEF EPSON,AM355=2:9600,TELTYP,100,100,100
```

This Epson printer may then be accessed as an output device by the file specification for any I/O operations requiring a specific device:

```
TRM:EPSON
```

To output directly to this device from BASIC, you would first open the device:

```
OPEN #file,"TRM:EPSON",OUTPUT
```

and then display data as follows:

```
PRINT #file,variable-list
```

The variable list may contain text and variables that you want to print as well as PRINT USING masks.

If you are planning on using terminals as I/O devices or to spool to them, it might save some space to include the TRM.DVR program in system memory during system startup. You would use the command

```
SYSTEM TRM.DVR[1,6]
```

before the last SYSTEM command in the system initialization command file.

Note that if you are using AlphaBASIC, AlphaVUE, or any other program that does its own memory management not consistent with the normal AMOS memory module scheme, you must pre-load TRM.DVR before using such a program to access TRM:. For example:

```
UNYANK TRM:TI810
```

THE TERMINAL CONTROL BLOCK

The terminal control block (TCB) is the primary data structure associated with the terminal service system. Because AMOS provides monitor calls to perform just about every possible terminal related function, the internal format of the TCB is generally of no interest to the programmer or user. However, because terminal service is such an integral part of AMOS, an understanding of how it is implemented can benefit the programmer by providing better insight into the structure and philosophy of AMOS itself.

However, because the contents of the TCB change from AMOS release to AMOS release, and because the procedure for accessing the TCB is somewhat complex and vital to system integrity, we recommend that all access to TCB fields be done via the monitor calls provided for that purpose (which handle all necessary interlocking themselves) and that no direct access to the TCB be done at all. The information provided in the remainder of this section is provided for information purposes only.

You can access the TCB associated with the terminal connected to the current job by executing the code shown below. This code sets an index to the associated TCB so that you can inspect or modify the items within. You can access TCBs associated with other terminals via the TCBIDX call, described later.

```
JOBIDX   A6            ; index our JCB with A6
MOV      JOBTRM(A6),An ; index TCB with An
```

AMOS allows the TCB data structure to be shared among multiple processors in a single system. For this reason it is vital that proper inter-processor interlocking be done when accessing any field within the TCB. To provide for multi-processor coordination, each TCB contains an access semaphore which your program ***must*** acquire before assuming that any field within the TCB is valid, and before modifying the contents of a TCB. Failure to properly acquire and de-acquire this semaphore can lead to serious system problems.

To gain access to a TCB, use the following code sequence (the example assumes that the TCB in question is indexed by A5):

```
        SVLOK                  ; disable interrupts
10$:    TAS      T.SEM(A5)     ; try to get semaphore
        BNE      10$           ;   not available - wait
```

Once you have completed all TCB accesses, you can release the TCB with the sequence:

```
CLRB     T.SEM(A5)     ; release the semaphore
SVUNLK                 ; allow interrupts
```

Because interrupts must be disabled during TCB access, and because acquiring the semaphore denies all other processors access to the TCB, it is important to minimize the amount of time during which you hold the TCB. Failure to properly disable and enable interrupts, in the correct order, will lead to system deadlock situations. (Note that within interrupt routines, interrupts are already disabled, allowing you to eliminate the SVLOK and SVUNLK calls.)

The following sections describe each of the fields with the TCB. Each field is accessed as an offset from a TCB pointer. The symbolic name for each field is defined in the symbol file TRM.UNV.

T.STS - The Terminal Status Word

This 16-bit status field is usually the only item within the TCB with which you need be concerned. It has certain flags in it that you may modify to alter the operation of your terminal calls.

To avoid conflicts with future changes in the terminal status word, do not modify the bits in the status terminal word directly: instead, use the TRMRST and TRMWST monitor calls, discussed in Chapter 7, which handle all interprocessor coordination for you.

The terminal status word has the following flag positions defined:

Symbol	Octal Value	Hex Value	Purpose
T$IMI	1	1	Set to force image mode input (see KBD call)
T$ECS	2	2	Set to suppress echoing of input characters
T$LCL	4	4	Set if terminal has local echoing (half-duplex)
T$DAT	10	8	Set to engage data mode to allow complete data transparency on input and output ^C, nulls, and 8-bit characters are all passed through without special processing.
T$ILC	20	10	Set to allow lower-case input (disables conversion)
T$XLT	40	20	Set to allow multi-key sequences for function key translation
T$NFK	100	40	Disables all function key processing—overrides T$XLT
T$OIP	200	80	Set if output is in progress (internal flag only)
T$LED	400	100	Set if monitor line editor is in use by this terminal
T$ASN	1000	200	Set if this terminal port is assigned
T$DIS	2000	400	Set if this terminal port is disabled
T$VLD	4000	800	Set if T.POO field contains a valid value
T$LDT	10000	1000	Set if in special line editor mode
T$EXT	20000	2000	Set if program wants extended (8-bit) characters
T$OSP	40000	4000	Output has been suspended (XOFF)
T$JLVL	100000	8000	Output data processing to be done at job level

The monitor resets the TIMI, TECS, T$DAT, and T$ILC bits in the terminal status word each time your program exits back to AMOS command mode, thereby restoring normal terminal operation regardless of program operation.

T.IDV - Pointer to Interface Driver

This 32-bit field contains a pointer to the interface driver (IDV) associated with this terminal. This pointer points directly to the first word of the interface driver. The name of the interface driver (packed RAD50) is contained in the longword at an offset of -4 from this pointer.

T.IHW - Interface Hardware Address

This 32-bit field contains the base address of the interface hardware associated with this terminal.

T.IHM - Interface Hardware Address Modifier

This 32-bit field contains the offset from the base hardware address for the particular interface port used by this terminal. This is often referred to as the ***port number***. The exact meaning of this value is interface dependent.

T.TDV - Pointer to Terminal Driver

This 32-bit field contains a pointer to the terminal driver (TDV) associated with this terminal. This pointer points directly to the first word of the terminal driver. The name of the terminal driver (packed RAD50) is contained in the longword at an offset of -4 from this pointer.

T.ICC - Input Character Count

This 32-bit field contains the number of input characters currently awaiting processing. This field is used as an offset into the input buffer indexed by T.IBF to determine where to place the next incoming character.

To check if input is available, use the TCKI monitor call, rather than checking this field.

T.ECC - Echo Character Count

This 32-bit field contains the number of characters that have been echoed since the last break character (carriage return, etc.).

T.BCC - Break Character Count

This 32-bit field contains the number of characters that have been input since the last break character (carriage return, etc.).

T.IBF - Input Buffer Address

This 32-bit pointer indexes the input buffer for this terminal. The T.ICC field is used as an index into this buffer.

T.IBS - Input Buffer Size

This 32-bit field contains the size of the input buffer indexed by T.IBF.

The size contained in this field is set by the TRMDEF command defining this terminal during the system initialization command file.

T.OQX - Output Queue Index

This 32-bit field contains a pointer to a linked list of queue blocks used to describe the current output queue. If no output is pending, this field will be zero.

T.OBX - Output Buffer Index

This 32-bit field is an offset into the currently active output buffer (pointed to by T.OBF). This offset is used to determine where to place the next output character.

T.OBF - Output Buffer Address

This 32-bit field points to the currently active output buffer. AMOS maintains two output buffers which it alternately fills. This field points to the currently active output buffer.

T.OBS - Output Buffer Size

This 32-bit field contains the size of the output buffer indexed by T.OBF.

The size contained in this field is set by the TRMDEF command defining this terminal during the system initialization command file.

T.OBD - Output Buffer XOR Difference

This 32-bit field contains the result of exclusive ORing the addresses of the two output buffers. This value is used as a very quick method of alternating between two buffers. (By exclusive ORing the contents of T.OBD with the address of the current buffer, you will end up with the address of the other buffer, without having to worry about which is which.)

T.POB - Beginning Output Position

This 16-bit field contains the column number of the position where the current input stream started. It is used primarily for Control-U processing.

T.POO - Current Output Position

This 16-bit field contains the column number of the current cursor position. This field is primarily used for Control-U processing. (T.POO minus T.POB yields the number of characters to echo.)

This field is not updated by TCRT commands such as cursor positioning, thus making its contents valid only in a line input situation. A program can determine if the contents of T.POO are valid by checking the T$VLD flag in the terminal status word (T.STS) which will be set if T.POO is valid. T.POO is reset whenever a carriage return is processed, thereby resetting the value and setting the T$VLD flag.

T.LCH - Last Character Input

The low-order byte of this 16-bit field contains the value of the last character to have been input from this terminal.

T.JLK - Attached JCB Pointer

This 32-bit field contains a pointer to the JCB of the job that this terminal is attached to. If the terminal is not attached to any job, this field will contain a zero.

T.ILB - Input Line Buffer Address

This 32-bit field contains a pointer to the input line buffer. It is this buffer that A2 will index upon completion of a line mode KBD monitor call.

T.ILS - Input Line Buffer Size

This 32-bit field contains the size of the input line buffer pointed to by T.ILB. This size is the maximum number of characters that can be entered on a single line, regardless of the size of the internal input buffers. Any attempt to enter more characters will result in a bell being echoed back.

The size contained in this field is set by the TRMDEF command defining this terminal during the system initialization command file.

T.IMP - Pointer to Terminal Driver Impure Area

This 32-bit field contains a pointer to the impure space allocated by the terminal driver (TDV) in use by this terminal. Impure space is allocated at TRMDEF time within the system initialization command file. It is up to the terminal driver to determine the amount of impure space needed.

If no impure space is used by the terminal driver, this field will contain a zero.

T.BAU - Selected Baud Rate

This 16-bit field contains the transmission baud rate currently selected for this terminal port. The value is taken from the table below:

Octal Value	Hex Value	Setting	Octal Value	Hex Value	Setting
0	0	50 baud	12	A	2000 baud
1	1	75 baud	13	B	2400 baud
2	2	110 baud	14	C	3600 baud
3	3	134.5 baud	15	D	4800 baud
4	4	150 baud	16	E	7200 baud
5	5	200 baud	17	F	9600 baud
6	6	300 baud	20	10	19200 baud
7	7	600 baud	21	11	38400 baud
10	8	1200 baud	22	12	57600 baud
11	9	1800 baud			

Note that not all interfaces support all baud rates. Refer to the documentation specific to the interface in question to see if a particular baud rate is supported.

The baud rate for an interface may be changed via the COMINT monitor call.

T.MLT - Multiple Character Queue Link

This 32-bit field contains a pointer to the multi-character input queue, if multi-character input is currently active. Multi-character input is used for the detection of function keys. This field is used, within AMOS and within terminal drivers that support multi-character input handling, as a method of keeping track of what characters have been seen during the multi-character input time period.

T.SEM - Multi-processor Interlock Semaphore

This 16-bit field is used as a semaphore to protect the integrity of the TCB during multi-processor operation. This semaphore must be used by all software accessing the TCB.

T.MRP - Modem Command Response

This 16-bit field is used to return the status of a requested modem driver call. Because the completion of a modem driver request can be asynchronous to the request itself, the status is returned here.

The flags returned in this field are defined as follows:

Symbol	Value	Meaning
T$MRO	1	Modem response was OK
T$MRF	2	Modem response failed

T.LED - Line Editor Dispatch

This 32-bit field contains a pointer to the monitor line editor routine. If the monitor line editor is not enabled for this terminal, this field will contain a zero.

T.FXT - Function Key Translation Pointer

This 32-bit field contains a pointer to the function key translation table used by the monitor line editor. If the monitor line editor is not enabled for this terminal, or if no function key translation table was found for the terminal driver associated with this TCB, this field will contain a zero.

T.INC - Input Character Routine Address

This 32-bit field contains a pointer to the user supplied routine which will be handling input characters at the interrupt level. The routine indexed by this field will be called each time an input character arrives at the interface driver.

This field should not be accessed directly, but should be set via the COMINT monitor call. More information on the use of this field can be found in Chapter 16.

This field will be zero if no interrupt routine is currently active.

T.OTC - Output Character Routine Address

This 32-bit field contains a pointer to the user supplied routine which will be handling output characters at the interrupt level. The routine indexed by this field will be called each time the interface driver is ready to output another character.

This field should not be accessed directly, but should be set via the COMINT monitor call. More information on the use of this field can be found in Chapter 16.

This field will be zero if no interrupt routine is currently active.

T.EXC - Exception Routine Address

This 32-bit field contains a pointer to the user supplied routine which will be handling exception conditions at the interrupt level. The routine indexed by this field will be called each time an exception is detected by the interface driver.

This field should not be accessed directly, but should be set via the COMINT monitor call. More information on the use of this field can be found in Chapter 16.

This field will be zero if no interrupt routine is currently active.

T.MDV - Modem Driver Pointer

This 32-bit field contains a pointer to the modem driver (MDV) associated with this TCB. Modem drivers are specified within the TRMDEF command in the system initialization command file.

If no modem driver is associated with this TCB, this field will contain a zero.

T.ASJ - Pointer to Assigning Job

This 32-bit field contains a pointer to JCB of the job that currently has this TCB assigned. If this TCB is not assigned, this field will contain a zero.

TCBs are assigned through the use of the MDREQ monitor call. For a TCB to be assigned, it must have an associated modem driver and not be attached to any job.

T.TCX - Pointer to TCRT Translation Table

This 32-bit field contains a pointer to the currently active TCRT translation table for this terminal. When this field is non-zero, all TCRT functions sent to this terminal will be translated through this table before actually forwarding the TCRT request to the terminal driver. If this field is zero, no translation is performed.

The purpose of this translation table is to allow software that does not conform to the terminal usage standards to be used without needing to correct the non-conforming software.

The translation table indexed by this field consists of 256 words. All TCRT codes with a high byte of -1 are passed through this table by using the low-order byte of the TCRT command as a word index into the table, selecting one of the 256 possible values. If the contents of the translation table at that address are zero, no translation is performed. If the contents are non-zero, the word from the translation table is taken to be the new TCRT value. If this new value contains a -1 in the high byte, it is also passed through the translation table, with the process repeating itself until a zero entry in the table is reached. This allows multiple translation passes through the table.

This field is set to zero by the AMOS EXIT call when no further input is available from command files or from the JOBCMD forced command (i.e., when AMOS will prompt the user for input, rather than getting it from any other source).

T.MBF - Pointer to Modem Driver Impure Area

This 32-bit field contains a pointer to the impure space allocated by the modem driver (MDV) in use by this terminal. Impure space is allocated at TRMDEF time within the system initialization command file. It is up to the modem driver to determine the amount of impure space needed.

If no impure space is used by the modem driver, or if no modem driver is associated with this terminal, this field will contain a zero.

T.OBE - End of Current Output Buffer

This 32-bit field points to the end of the current output buffer. For AMOS internal use only.

T.OWAT - Output Wait Chain

This 32-bit field points to the output wait chain.

T.SIS - Software Interrupt Structure

This 32-bit field points to the software interrupt structure associated with this terminal.

T.SIV - Software Interrupt Vector

This 32-bit field points to the software interrupt vector table associated with this terminal.

T.STSZ - Second Terminal Status Word

This 16-bit field contains the Second Terminal Status Word:

Symbol	Value	Meaning
T$OCP	1	Reserved for internal AMOS use
T$IACT	2	Interface is active

Appendix C
System Communication Area

The area in monitor memory starting at location 2000_8 is called the system communication area. This area is used to store data that define the current system state and configuration. These items reflect global, system-wide data that apply to the system as a whole, rather than to a single job. The symbols used in this appendix are defined in the symbol file SYS and SYSSYM.

The data structures contained in the system communication area are briefly defined here for those who wish to reference them; however, you should rarely change these fields, and you must exercise extreme caution if you do. Arbitrarily modifying almost any of these fields will most likely result in system failure.

You should make all references to these parameters symbolically in the absolute addressing mode. For example, you would use the instruction MOV JOBTBL,A0 to set the base of the user job table into address register A0.

SYSTEM - SYSTEM ATTRIBUTES WORD

This longword contains the following system attribute and status flags:

Symbol	Octal Value	Hex Value	Meaning
SY$UP	1	1	Final SYSTEM command has been given
SY$LNK	2	2	Obsolete
SY$CPA	4	4	System is running on an AM-1000
SY$NET	10	8	System is on a network
SY$LOK	20	10	The system is running LOKSER
SY$M10	40	20	Central processor is a 68010
SY$CPB	100	40	CPU is running on a VME bus based system
SY$VNT	200	80	System connected to a video network
SY$TST	400	100	System is in "test" mode. This flag is set via the SET TEST command.
SY$CPC	1000	200	System is running on an AM-170 ELS system
SY$LNG	2000	400	System has language definition tables installed
SY$NBP	4000	800	System contains "new" boot PROMs. For internal use only.
SY$HFP	10000	1000	The system contains hardware floating point
SY$M20	20000	2000	Central processor is a 68020
SY$SIS	40000	4000	System supports the software interrupt system
SY$M30	100000	8000	Central processor is a 68030
SY$EXD	200000	10000	System supports extended format directories
SY$NEL	1000000	40000	System is running on an AM-113 AMPC system
SY$CPD	2000000	80000	System is running on an AM-2000M system
SY$TBX	4000000	100000	System has programmer's toolbox routines installed
SY$EXT	10000000	200000	System supports 8-bit character sets.
SY$CAR	100000000	1000000	System is running from the cartridge disk
SY$CPE	200000000	2000000	System is running on an AM-1400.
SY$CPF	400000000	4000000	System is running on an AM-1600.
SY$CPG	1000000000	8000000	System is running on an AM-3000M.
SY$M40	2000000000	10000000	System is running on 68040 processor.
SY$M60	4000000000	20000000	System is running on 68060 processor.

DEVTBL - ADDRESS OF THE DEVICE TABLE

The DEVTBL program in the system initialization command file sets up this longword to contain the absolute address of the device table in monitor memory.

DDBCHN - ACTIVE DDB CHAIN

This is the base of the active DDB chain for interrupt driven routines. The file service routines sets it up and alters it as new IO DDB's are queued for transfer requests. It goes to zero each time there are no requests pending, and is not used for non-interrupt driven devices and certain intelligent IO controllers.

MEMBAS & MEMEND - USER MEMORY POINTERS

These two longwords 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. AMOS sets it up when the monitor first starts up.

SYSBAS - BASE OF SYSTEM MEMORY

This is the address of the system memory area, which contains any user programs set up by the SYSTEM command in the system initialization command file. It is zero if no system memory area exists. The format of this area is the same as for user memory modules.

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 JOBS command in the system initialization command file. For a complete description of the job table and JCB entries, refer to Chapter 2, “Job Scheduling and Control System.”

JOBCUR - JCB ADDRESS OF THE CURRENT JOB

This longword 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 always points to your own JCB, since you must be running to reference this word. Within an interrupt service routine, JOBCUR may contain a zero or the JCB address of a job. Only the AMOS's job scheduler updates JOBCUR.

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, you won't have to reassemble the programs which scan the job table since they get the JCB size dynamically from JOBESZ. This includes routines within the monitor itself.

LOKSEM - RECORD LOCKING SEMAPHORE

This is a two longword field used as a semaphore to force record locking requests to be serial.

TIMQUE - THE TIMER QUEUE

This longword points to the first entry in the timer queue. If no items are in the timer queue, this entry is zero.

WEREUP - SYSTEM BOOT INDICATOR

If this location contains $F321_{16}$, then the system has been booted at least once.

SPXSAV - STACK POINTER SAVE LOCATION

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

SPXINT - INTERNAL STACK

This is the address of the internal work stack used for processing timer interrupts. The initial load routine sets it up, and the timer interrupt processor uses it.

LPTQUE - LINE PRINTER SPOOLER QUEUE

This is the dynamic link address to the base of the queue used by the memory based line printer spooler (LPTSPL).

TRMDFC - BASE OF TERMINAL DEFINITION TABLE

This is the link to the base of a linked list of terminal control blocks (TCBs). There is one TCB for each terminal defined at system startup by a TRMDEF statement in the system initialization command file, or subsequently created via job spawning.

TRMIDC - ADDRESS OF FIRST INTERFACE DRIVER

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

TRMTDC - ADDRESS OF FIRST TERMINAL DRIVER

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

LOKADR - LOKSER ADDRESS

Not used under AMOS 2.0.

UPTIME - TIME AND DATE OF LAST SYSTEM RESET

This longword contains the packed time and date of the last system reset. By subtracting this value from the current time and date, you can determine the length of time the system has been up. Bits 0-16 contain the time of day (in internal format) that the system was booted. Bits 17-31 contain the date the system was booted, where the date is stored as the number of days since January 1, 1980. You can convert the packed date to standard internal date format by adding 2444240 (decimal). You can also use the $OTCON subroutine described in Appendix D to unpack and display this field.

JLKCNT - THE JLOCK NESTING COUNTER

This longword is the counter for nesting JLOCK monitor calls. It is zero when no JLOCKs are active, non-zero when at least one is active. Note that only the user executing the JLOCK will ever see this non-zero.

WHYBOT - REASON FOR LAST SYSTEM REBOOT

This two-byte field contains a code specifying the reason for the last system halt. The contents of WHYBOT are only valid if the high order byte contains 125_8. If this value is not present the low-order byte is meaningless. If it is present, the low-order byte is interpreted as follows:

Octal Value	Hex Value	Meaning
4	4	The system ran out of queue blocks
376	FE	The system was rebooted via the REBOOT program
377	FF	The system was rebooted via the MONTST program

NETTBL - POINTER TO NETWORK LIST

This longword field contains a pointer to the list of defined networks. If no networks are defined, this field will be a zero.

NETBUF - POINTER TO NETWORK BUFFER AREA

This longword field points to the buffer space allocated for use by the ITC system (via the MSGINI program). If no buffer space has been allocated, this field will be zero.

DCACHE - DISK CACHE DISPATCH POINTER

This 32-bit field contains a pointer to the disk cache handler address. If the disk cache is not active, this field will contain the address of a routine which simply sets the appropriate flags and returns.

SYSLNG - DEFAULT SYSTEM LANGUAGE

This longword field contains a pointer to the default system language definition table which was loaded at MONGEN time.

HLDADR - HEAD LOAD TIMER ADDRESS

This is the low-order 16 bits of the I/O port address for the floppy disk controller currently using the head load timer.

TMRLOK - TIMER INTERRUPT FLAG

One word field that is used internally to flag that the operating system is currently working within a timer interrupt. This is necessary to avoid improper nesting of interrupts. This field must never be modified.

DRVTRK - THE DRIVE/TRACK TABLE

DRVTRK is a 4-byte block that stores 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.

HLDTIM - HEAD LOAD TIMER COUNT

This is the amount of time (number of ticks) to wait between the last access to the floppy disk and turning off the motor.

SCKTLS - POINTER TO LIST OF ASSIGNED SOCKETS

This 32-bit field contains a pointer to the list of currently assigned ITC sockets. If no sockets are assigned, this field will contain a zero.

ZSYDSK - ADDRESS OF SYSTEM DISK DRIVER

This longword contains the base address of the system disk driver within the monitor. MONGEN uses it to overlay the disk driver with another one when changing the resident disk type.

SYSLNK - SYSTEM LINK COMMUNICATIONS

Reserved for future use.

SCLKON - SCHEDULER CLOCK ENABLED FLAG

This 16-bit field contains a flag used by the AMOS scheduler and timer routines to flag that the scheduler clock is enabled.

QFREE - QUEUE SYSTEM CONTROL

QFREE consists of two longwords, 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, "Monitor Queue System Calls.”

MEMQUE - SYSTEM MEMORY QUEUE POINTER

Reserved for future use.

SYSUFD - SYS: UFD POINTER

This longword contains the block number of the first directory block for SYS: (DSK0:[1,4]). It is stored here to avoid reading the MFD on each access to DSK0:[1,4]. The DSKMNT and DSKUMT monitor calls update this block number.

DVRUFD - DVR: UFD POINTER

This longword contains the block number of the first directory block for DVR: (DSK0:[1,6]). It is stored here to avoid reading the MFD on each access to DSK0:[1,6]. The DSKMNT and DSKUMT monitor calls update this block number.

CMDUFD - CMD: UFD POINTER

This longword contains the block number of the first directory block for CMD: (DSK0:[2,2]). It is stored here to avoid reading the MFD on each access to DSK0:[2,2]. The DSKMNT and DSKUMT monitor calls update this block number.

BASUFD - BAS: UFD POINTER

This longword contains the block number of the first directory block for BAS: (DSK0:[7,6]). It is stored here to avoid reading the MFD on each access to DSK0:[7,6]. The DSKMNT and DSKUMT monitor calls update this block number.

ERSATZ - ACCESSES ERSATZ DEVICE TABLE

This longword points to the start of the ersatz device table. The ERSATZ program builds a table in the operating system containing the defined ersatz device specifications. If this longword is 0, no ersatz devices are defined, otherwise it points to a table formatted as:

Symbol	Size	Meaning
EZ.NAM	4 bytes	Ersatz device name, packed RAD50 or 0 to terminate the ersatz table
EZ.CPU	4 bytes	Default CPU specification
EZ.DEV	2 bytes	Default device specification
EZ.UNT	2 bytes	Default unit number
EZ.FIL	4 bytes	Default filename
EZ.EXT	2 bytes	Default extension
EZ.PPN	2 bytes	Default PPN

SYSNAM - NAME OF SYSTEM MONITOR

This 16-byte field contains the name of the system monitor in ASCII, terminated by a null. In the standard release the name of the system is pre-defined as either "AMOS/L" or "AMOS/32." Programs which must output the name of the system (STAT, SYSTAT, etc.) should display this field rather than using an embedded text string.

AMGDSP - AMIGOS DISPATCH VECTOR

This 32-bit field contains a pointer to the entry point of AMIGOS. If AMIGOS is not installed, this pointer will index a routine that returns an error in standard AMIGOS format.

SCHSEM - SCHEDULER SEMAPHORE

This single byte field contains a TAS style semaphore used to protect the scheduler data structures during multi-processor access.

QUESEM - QUEUE SYSTEM SEMAPHORE

This single byte field contains a TAS style semaphore used to protect the queue system data structures during multi-processor access.

RIOQUE - RECORD IO QUEUE

This 32-bit pointer indexes a queue used to coordinate physical block access within the record IO system.

LEDDSP - LINE EDITOR DISPATCH VECTOR

This 32-bit pointer indexes the line editor routine. If the line editor is not active, this field will contain a zero.

TRMFXC - FUNCTION KEY TRANSLATION TABLE CHAIN

This 32-bit field contains a pointer to a linked list of terminal function key translation tables for use by the line editor. If no function key translation tables are available, this field will contain a zero.

TRMMDC - MODEM DRIVER CHAIN

This 32-bit field contains a pointer to a linked list of modem drivers available on the system. If no modem drivers are available, this field will contain a zero.

HRBCMD - HERBIE COMMAND BLOCK POINTER

This longword field indexes the command block initially used by Herbie type controllers.

PRESEM - SEMAPHORE TO PROTECT SCHEDULER FIELDS

This byte field contains a TAS style semaphore used to protect the next two fields during multi-processor access. Failure to properly use this semaphore during access to these two fields will cause certain system failure.

PREFLG - SCHEDULER FLAGS

This 16-bit field contains flags that may be passed to the scheduler during certain internal operations.

PREJCB - SCHEDULER JCB ADDRESS

This 16-bit field contains a JCB address that may be passed to the scheduler during certain internal operations.

NULTMR - NULL TIMER ROUTINE POINTER

This longword pointer indexes the null timer routine within AMOS. Originally used to allow for the removal of timer requests from a queue, this field has now been obsoleted by the more general DQTIMR monitor call.

FPNPTR - POINTER TO ISAM FILE TABLE

This longword field contains a pointer to a list of open ISAM files. This field should only be used by the ISAM file processor itself.

HRBERR - POINTER TO HERBIE ERROR HANDLER

This 32-bit field contains a pointer to the fatal error handler supplied by AMOS to Herbie type intelligent controllers.

MTSRES - MONTST RESET CHAIN

This 32-bit field contains a pointer to a linked list of routines that will be called when the MONTST program is executed. Each routine which wishes to be called must link itself into this list. MONTST will then call each routine (at the address immediately following the link) before starting the actual reboot process. This provides "advance notice" to software of the impending reboot, allowing it to reset hardware to a known state. All registers must be preserved within this routine.

LOKFLH - RECORD LOCK STRUCTURE POINTER

This 32-bit field contains a pointer into the record locking structure within the AMOS file system.

USMEXT - USAM EXIT HANDLER DISPATCH VECTOR

This 32-bit field contains a pointer to USAM's EXIT handler. This routine is called each time an EXIT is performed.

SVCPTR - POINTER TO SUPERVISOR CALL DISPATCH TABLE

This 32-bit field contains a pointer to AMOS's internal supervisor call dispatch table. It is provided for the use of the Herbie intelligent controllers. Because its format is subject to change, we do not recommend that other software make use of this pointer.

RFDVEC - VDK VECTOR

This 32-bit field contains a pointer to the VDK virtual disk routines, if installed. These routines should not be called directly.

RFDPTR - POINTER TO VDK IMPURE SPACE

This 32-bit field contains a pointer to the impure space used by the VDK virtual disk routines. This impure space should not be directly accessed.

PLKJCB - POINTER TO PARENT JOB OWNING PLOCK RESOURCE

This 32-bit field contains a pointer to the JCB of the parent job that currently owns the PLOCK resource. If no PLOCK is in effect, this field will be zero.

PLKCNT - PLOCK NESTING COUNT

This 32-bit field contains the number of PLOCKs that are currently in effect. Zero when PLOCK is not active, this field is used to properly nest and un-nest PLOCK calls.

TIMIDX - POINTER TO INTERNAL TIMER ROUTINES

This 32-bit field is used internally to support a variety of hardware timers. It should not be otherwise accessed.

ESPVEC - POINTER TO ESP ROUTINES

This 32-bit field contains a pointer to the control routines used by the ESP screen processor.

DDBSEM - DDBCHN ACCESS SEMAPHORE

This 8-bit field is used as a TAS style semaphore to protect the DDBCHN resource. It should never be accessed directly.

DDBSM2 - DDBCHN ACCESS SEMAPHORE

This 8-bit field is used as a TAS style semaphore to protect the DDBCHN resource. It should never be accessed directly.

HCFLAG - ENABLE HERBIE CACHING

This 8-bit field is flag that enables caching of Herbie-style intelligent controllers. It is used internally by Alpha Micro, and should not be used by user programs.

SYSCOF - POINTER TO LIST OF CURRENTLY OPEN OBJECT FILES

This 32-bit field contains a pointer to a list of the currently open object files.

TBXDSP - TOOLBOX DISPATCH VECTOR

This 32-bit field contains a pointer to ESP's toolbox program dispatch table.

RPCDSP - RPC DISPATCH VECTOR

This 32-bit field contains a pointer to the RPC dispatch table.

EXTDSP - EXTENSION DISPATCH VECTOR

This 32-bit field is reserved for future use.

QXFRAD - PHYSICAL DISK TRANSFER SYSTEM

This 32-bit field contains a pointer to the physical disk transfer queuing entry.

SCHEDW - FOR WATCHR PROGRAM

This 32-bit field is reserved for use of the WATCHR program.

VTJOBT - VTSER SPAWNED JOBS

This field contains a pointer to the VTSER spawned job table.

ETHZON - ETHERNET COMMUNICATIONS AREA

This 32-bit field contains a pointer to the Ethernet communications area used by AlphaNET. This field must never be modified.

TTYPTR - TTYSI POINTER

This 32-bit field contains a pointer to the TTYSI service routine. This field is used internally by AMOS and must never be modified.

XTABLE - X.25 TABLE POINTER

This 32-bit field contains a pointer to a table used by the X.25 communications software. It must never be modified.

UNXVEC - UNIX ACCESS VECTOR

This 32-bit field is reserved for future use.

OSI4VC - OSI LEVEL 4 VECTOR

This 32-bit field is used internally by the OSI Layer 4 communications software. This field should never be modified.

NETVEC - NETFAM VECTOR TABLE POINTER

This 32-bit field is used internally by the Network File Access Method (NETFAM) and should never be modified.

SEM522 - AM-522 INTERRUPT PENDING SEMAPHORE

This 8-bit field is used by the AM-522 disk controller to control interrupt access. This field must never be modified.

VEC522 - AM-522 INTERRUPT SERVICE VECTOR

This 8-bit field is used to supply the appropriate interrupt vector to the AM-522 disk controller. This field must never be modified.

FLEVEL - SYSTEM FEATURE LEVEL

This 32-bit field is used internally to determine the feature level of AMOS.

SYSTEM1 - SYSTEM1 ATTRIBUTE WORD

This 32-bit field is reserved for AMOS use only.

MSGBFE - MSGINI BUFFER

This field points to the end of the buffer defined by MSGINI.LIT.

CPUTYP - CPU TYPE

This field contains the CPU board type. It is set by INITIA.

SCZDSP

This field is reserved for AMOS use only.

DIAG 01

This field is reserved for AMOS use only.

DIAG 02

This field is reserved for AMOS use only.

DIAG 03

This field is reserved for AMOS use only.

EMAILV

This field is reserved for AMOS use only.

JRC.ADDR

This field is reserved for AMOS use only.

RTCIDX

This field is reserved for AMOS use only.

UMEMIDX

This field is reserved for AMOS use only.

TAMEV

This field is reserved for AMOS use only.

RSCPM

This longword field contains information about PROMPT.SYS.

FP060

This field is reserved for AMOS use only.

Appendix D Standard System Library Routines

In addition to the monitor calls described in this manual, AMOS also includes a collection of standard system subroutines. This collection, contained in the file DSK0:SYSLIB.LIB[7,7] contains various routines to perform common system operations.

For convenience, whenever the linking loader, LNKLIT, has completed loading the files you have specified, but still has unresolved global symbols, it automatically searches SYSLIB.LIB. Thus, to include one of these system routines in your program, simply specify the routine name (such as $ODTIM) as an external within your source program. LNKLIT will take care of the rest.

To make updating this appendix easier, each subroutine appears on a separate page.

The subroutines included in this appendix are:

$ADPPN	$ADPPNX*	$BBCHK	$CHPPN
$CHPPNX*	$CMDER	$CPUPOL*	$DITOS*
$DLPPN	$DLPPNX*	$DSTOI	$ERPPN
$ERPPNX*	$FLSET	$FNPPN	$FNPPNX*
$FNUSR*	$GTARG	$HSHFL*	$IDTIM
$IDTIMX*	$INMFD	$INMFDX*	$KILPF
$MSGLOG**	$NETED	$ODTIM	$ODTM2
$OTCON	$OTCPU	$PAKDT**	$PAMFD
$PAMFDX*	$SEND*	$SPLFL	$STFRM
$SYSID	$UNPDT	$UPDSW	$YESNO

*: These routines are not in the AMOS 1.X library, but are supported.

**: These routines are not in the AMOS 1.X library, and are not supported.

$ADPPN

Add a PPN to the MFD

AMOS 2.X only

The $ADPPN routine provides a standard method of adding new PPN entries to the MFD. To add a new PPN, you simply specify the PPN to be added, the password, the DDB and buffer to be used, and then call $ADPPN. You must be logged into [1,2] to use this subroutine.

$ADPPN works only with AMOS 2.X. If you want to support both AMOS 2.X and AMOS 1.X, use $ADPPNX instead.

Before calling $ADPPN, the following registers must be set up:

D1	Contains the PPN to be added in the low-order half of the register.
D2	Contains the password to be used for the new PPN, packed in RAD50 format.
A2	Points to an INITed DDB to be used for the add operation.

$ADPPN returns with the following:

D0 Contains the completion code:

0	PPN was added successfully
-1	The error code is contained in the DDB (D.ERR(A2))
-2	The MFD is damaged in some way
1	You are not logged into [1,2]
2	The specified PPN already exists
3	The specified PPN is not valid

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

All registers except D0, D6, D7, and A6 are preserved. The directory is locked (via DSKDRL) during execution of this routine. This routine sets the D$ERC bit in the DDB flags byte.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$ADPPNX

Add a PPN to the MFD

AMOS 2.X or 1.X

The $ADPPNX routine provides a standard method of adding new PPN entries to the MFD under either AMOS 2.X or 1.X. At run time, $ADPPNX checks the current operating system, then executes the proper version of $ADPPN.

To add a new PPN, you simply specify the PPN to be added, the password, the DDB and buffer to be used, and then call $ADPPNX. You must be logged into [1,2] to use this subroutine.

Before calling $ADPPNX, the following registers must be set up:

D1	Contains the PPN to be added in the low-order half of the register.
D2	Contains the password to be used for the new PPN, packed in RAD50 format.
A2	Points to an INITed DDB to be used for the add operation.

$ADPPNX returns with the following:

D0 Contains the completion code:

0	PPN was added successfully
-1	The error code is contained in the DDB (D.ERR(A2))
-2	The MFD is damaged in some way
1	You are not logged into [1,2]
2	The specified PPN already exists
3	The specified PPN is not valid

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

All registers except D0, D6, D7, and A6 are preserved. The directory is locked (via DSKDRL) during execution of this routine. This routine sets the D$ERC bit in the DDB flags byte.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$BBCHK

Check for BADBLK.SYS[1,2]

The $BBCHK routine provides a standard method for checking to see if the file you are accessing on a physical drive is the BADBLK.SYS[1,2] file. First, the routine checks for the filespec BADBLK.SYS[1,2], then it checks if the device in the DDB is in the device table. Next, it checks if the device is the first logical unit on a physical device, and if it has an alternate track table. If all of these conditions have been met, the following error message will be output:

```
%Bypassing BADBLK.SYS[1,2]
     BADBLK.SYS exists to prevent bad blocks
     on a device from being allocated, and
     should never be directly accessed.
```

Before calling $BBCHK, the following register must be set up:

A4	Points to a DDB specifying the file to be checked.

Upon return from $BBCHK the Z-flag is set if the file is BADBLK.SYS[1,2], and reset if it is not.

All registers except D6, A0, and A6 are preserved. The error message that is output is under the control of the D$BYP bit in the DDB flags byte.

$CHPPN

Change a PPN’s Password

AMOS 2.X only

The $CHPPN routine provides a standard method of changing the password of a PPN in the MFD. To change a password, you simply specify the PPN to be changed, the new password, the DDB and buffer to be used, and then call $CHPPN. You may only change your own password unless you are logged into [1,2].

$CHPPN works only with AMOS 2.X. If you want to support both AMOS 2.X and AMOS 1.X, use $CHPPNX instead.

Before calling $CHPPN, the following registers must be set up:

D1	Contains the PPN to be changed in the low-order half of the register.
D2	Contains the new password to be used for the PPN, packed in RAD50 format.
A2	Points to an INITed DDB to be used for the add operation.

$CHPPN returns with the following:

D0 Contains the completion code:

0	Password was changed successfully
-1	The error code is contained in the DDB (D.ERR(A2))
-2	The MFD is damaged in some way
1	You are not logged into [1,2] and you tried to change another PPN's password
4	The specified PPN was not found in the MFD

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

All registers, except D0, D6, D7, and A6 are preserved. The directory is locked (via DSKDRL) during execution of this routine. This routine sets the D$ERC bit in the DDB flags byte.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$CHPPNX

Change a PPN's Password

AMOS 2.X or 1.X

The $CHPPNX routine provides a standard way to change the password of a PPN in the MFD, under both AMOS 2.X and 1.X. At run time, it checks the operating system version and calls the proper version of $CHPPN.

To change a password, you simply specify the PPN to be changed, the new password, the DDB and buffer to be used, and then call $CHPPNX. You may only change your own password unless you are logged into [1,2].

Before calling $CHPPNX, the following registers must be set up:

D1	Contains the PPN to be changed in the low-order half of the register.
D2	Contains the new password to be used for the PPN, packed in RAD50 format.
A2	Points to an INITed DDB to be used for the add operation.

$CHPPNX returns with the following:

D0 Contains the completion code:

0	Password was changed successfully
-1	The error code is contained in the DDB (D.ERR(A2))
-2	The MFD is damaged in some way
1	You are not logged into [1,2] and you tried to change another PPN's password
4	The specified PPN was not found in the MFD

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

All registers, except D0, D6, D7, and A6 are preserved. The directory is locked (via DSKDRL) during execution of this routine. This routine sets the D$ERC bit in the DDB flags byte.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$CMDER

Output Error Message and Locator

The $CMDER routine provides a way of outputting an error message on the user's terminal, along with a caret (^) symbol locating the point in the command line where the error was detected. This is the routine used by COPY, DIR, etc. to point out the location of command line errors.

Before calling $CMDER, the following registers must be set up:

A1	Contains a pointer to the error message to be displayed. This error message must be terminated by a zero byte.
A2	Contains a pointer into the command line buffer at the location where the error occurred.

$CMDER preserves all registers (except D6, D7, and A6).

$CPUPOL

Determine Computer Type, Processor Type, and AMOS Version

The $CPUPOL routine provides a standard method of checking the computer for computer type, processor chip type, and AMOS version. You can check the computer type or processor type by using the symbols provided (listed below).

Before calling $CPUPOL, you do not need to set up any registers. Call the routine as follows:

```
      .
      .
              SEARCH   CPUSYM
              AUTOEXTERN
      .
      .
              CALL     $CPUPOL       ; Check computer type, processor type,
                                     ;   and AMOS version.
              BMI      XX$           ; AMOS/32 is running.
              BEQ      XY$           ; AMOS/L is running.
              BPL      XZ$           ; AMOS/LC is running.
```

$CPUPOL returns the following:

D0	System type flag (LWORD)
D1	CPU processor type flag (LWORD)
D2	AMOS release version code (BYTE)

Condition codes:

N-flag reset for AMOS/LC
Z-flag set for AMOS/L
N-flag set for AMOS/32

All registers other than D0, D1, D2, D6, D7, and A6 are preserved.

$CPUPOL Example:

```
              .
              .
              .
              SEARCH   CPUSYM
              AUTOEXTERN
              .
              .
              .
              CALL     $CPUPOL       ; Check computer type, processor type,
                                     ;    and AMOS version.
              BMI      10$           ; AMOS/32 running.
              BEQ      20$           ; AMOS/L running.
              TYPECR   <AMOS-LC is running!>
              BR       30$           ; Continue.
      $10:    TYPECR   <AMOS-32 is running!>
              BR       30$           ; Continue.
      20$:    TYPECR   <AMOS-L is running!>
      30$:    ...
```

Below are the flags you can use to determine system type, processor type, and AMOS version. These flags are defined in the file CPUSYM.M68 in DSK0:[7,7], and may be referenced through the universal file CPUSYM.UNV.

System Type Flags = D0 (LWORD)

Symbol	AMOS Type	AMOS System
AM113	AMOS/LC	AM-PC
AM130	AMOS/LC	AM-1400
AM134	AMOS/LC	AM-1400
AM135	AMOS/L	AM-1600
AM140	AMOS/32	AM-3000M
AM145	AMOS/32	AM-2000M
AM160	AMOS/L	S/100
AM167	AMOS/L	AM-1000
AM172	AMOS/L	Roadrunner '030
AM174	AMOS/L	Roadrunner '040
AM175	AMOS/L	AM-1500
AM177	AMOS/L	AM-1200
AM180	AMOS/32	AM-2000
AM185	AMOS/32	AM-3000
AM190	AMOS/32	AM-4000
AM319	AMOS/32	Eagle
FALCON	AMOS/LC	Falcon

Processor Type Flags = D1 (LWORD)

Symbol	Description
C68000	M68000 CPU
C68010	M68010 CPU
C68020	M68020 CPU
C68030	M68030 CPU
C68040	M68040 CPU
C68060	M68060 CPU

AMOS Release Version = D2 (BYTE)

Octal Value	Hex Value	Meaning
1	1	AMOS/LC Version
0	0	AMOS/L Version
377	FF	AMOS/32 Version

Convert Internal Format Date to Separated Format

The $DITOS routine provides a way of converting a date stored in internal format to one stored in separated format. (Also see "$DSTOI," in this appendix for information on converting a date stored in separated format to internal format.) Additional information on date formats may be found in Chapter 10 of this manual.

$DITOS will work correctly for dates from January 1, 1900 through December 31, 2155, inclusive.

Before calling $DITOS, the following register must be set up:

D7	Contains the date to be converted, in internal format.

Upon completion $DITOS returns the following:

D7	Contains the converted date, in separated format.

$DITOS assumes that the internal format date is in Julian format.

This routine preserves all registers (except D6, D7, and A6).

$DLPPN

Delete a PPN from the MFD

AMOS 2.X only

The $DLPPN routine provides a standard method of deleting PPN entries from the MFD. To delete a PPN, you simply specify the PPN to be deleted, and the DDB and buffer to be used. Then call $DLPPN. You must be logged into [1,2] to use this subroutine. The PPN to be deleted must not contain any disk files.

$DLPPN works only with AMOS 2.X. If you want to support both AMOS 2.X and AMOS 1.X, use $DLPPNX instead.

Before calling $DLPPN, the following registers must be set up:

D1	Contains the PPN to be deleted in the low-order half of the register.
A2	Points to an INITed DDB to be used for the add operation.

$DLPPN returns the following:

D0 Contains the completion code:

0	PPN was deleted successfully
-1	The error code is contained in the DDB (D.ERR(A2))
-2	The MFD is damaged in some way
1	You are not logged into [1,2]
2	The specified PPN already exists
3	The specified PPN is not valid
4	The specified PPN was not found in the MFD
5	The specified PPN contains files

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

All registers except D0, D6, D7, and A6 are preserved. The directory is locked (via DSKDRL) during execution of this routine. This routine sets the D$ERC bit in the DDB flags byte.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$DLPPNX

Delete a PPN from the MFD

AMOS 2.X or 1.X

The $DLPPNX routine provides a standard method of deleting PPN entries from the MFD. At run time, it checks the operating system version and calls the proper version of $DLPPN.

To delete a PPN, you simply specify the PPN to be deleted, and the DDB and buffer to be used. Then, call $DLPPNX. You must be logged into [1,2] to use this subroutine. The PPN to be deleted must not contain any disk files.

Before calling $DLPPNX, the following registers must be set up:

D1	Contains the PPN to be deleted in the low-order half of the register.
A2	Points to an INITed DDB to be used for the add operation.

$DLPPNX returns the following:

D0 Contains the completion code:

0	PPN was deleted successfully
-1	The error code is contained in the DDB (D.ERR(A2))
-2	The MFD is damaged in some way
1	You are not logged into [1,2]
2	The specified PPN already exists
3	The specified PPN is not valid
4	The specified PPN was not found in the MFD
5	The specified PPN contains files

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

All registers except D0, D6, D7, and A6 are preserved. The directory is locked (via DSKDRL) during execution of this routine. This routine sets the D$ERC bit in the DDB flags byte.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$DSTOI

Convert Separated Format Date to Internal Format

The $DSTOI routine provides a way of converting a date stored in separated format to one stored in internal format. Additional information on date formats may be found in Chapter 10 of this manual.

Before calling $DSTOI, the following register must be set up:

D7	Contains the date to be converted, in separated format.

Upon completion $DSTOI returns the following:

D7	Contains the converted date, in internal format.

$DSTOI will work correctly for dates from January 1, 1900 through December 31, 2155, inclusive.

This routine preserves all registers (except D6, D7, and A6).

$ERPPN

Process Errors in PPN Routines

AMOS 2.X only

The $ERPPN routine provides a standard method of processing errors in the $ADPPN, $CHPPN, $DLPPN, and $FNPPN routines. To process an error in any of these routines, and output the appropriate error message, the error code is passed in D0, and then $ERPPN is called. Normally, $ERPPN is called automatically by these routines. If you need to call it yourself, you must first set up register D0, which contains the error code to be processed. $ERPPN outputs the following:

0	The function was successful, no error message
-1	DDB error, the monitor will output the error message
-2	?Damaged MFD
-3	Invalid operation on extended disk
1	?Not logged into [1,2]
2	?Account already exists
3	?Illegal account PPN - format is P, P(P = octal 1 to 377)
4	?Account does not exist
5	?Account has files on it

$ERPPN works only with AMOS 2.X. If you want to support both AMOS 2.X and AMOS 1.X, use $ERPPNX instead.

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs. If a -1 or -2 error occurs, the N-flag is also set. All registers are preserved. Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$ERPPNX

Process Errors in PPN Routines

AMOS 2.X or 1.X

The $ERPPNX routine provides a standard method of processing errors in the $ADPPNX, $CHPPNX, $DLPPNX, and $FNPPNX routines. To process an error in any of these routines, and output the appropriate error message, the error code is passed in D0, and then $ERPPNX is called. Normally, $ERPPNX is called automatically by these routines. If you need to call it yourself, you must first set up register D0, which contains the error code to be processed. $ERPPNX outputs the following:

0	The function was successful, no error message
-1	DDB error, the monitor will output the error message
-2	?Damaged MFD
-3	Invalid operation on extended disk
1	?Not logged into [1,2]
2	?Account already exists
3	?Illegal account PPN - format is P, P(P = octal 1 to 377)
4	?Account does not exist
5	?Account has files on it

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs. If a -1 or -2 error occurs, the N-flag is also set. All registers are preserved. Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$FLSET

Locate Open AlphaBASIC File

The $FLSET routine provides a standard way of locating open files within an AlphaBASIC XCALL subroutine. Given a file channel number, it returns a pointer to the DDB referencing the file open on that channel.

Prior to calling $FLSET, set up the registers as follows:

D1	Contains the file channel number to search for.
A0	Contains the AlphaBASIC impure area pointer. This register is already set up when an XCALL routine is called by BASIC.

Upon a successful return from $FLSET, A2 indexes the file DDB and the Z-flag is set. If no file was open on the specified file channel, the Z-flag is cleared.

For further information on AlphaBASIC XCALL subroutines, see the *AlphaBASIC XCALL Subroutine User's Manual*.

$FNPPN

Find a PPN in the MFD

AMOS 2.X only

The $FNPPN routine has been superseded by the DIRSCH monitor call. $FNPPN does not properly handle extended format directories and is provided only for compatibility with older software. Use of $FNPPN in new software is discouraged.

The $FNPPN routine provides a standard method of locating PPN entries within the MFD. To find a PPN, you simply specify the PPN to be located, the DDB and buffer to be used, and the call $FNPPN.

$FNPPN works only with AMOS 2.X. If you want to support both AMOS 2.X and AMOS 1.X, use $FNPPNX instead.

Before calling $FNPPN, the following registers must be set up:

D1	Contains the PPN to be located in the low-order half of the register.
A2	Points to an INITed DDB to be used for the find operation.

$FNPPN returns the following:

D0 Contains the completion code:

0	PPN was located successfully
-1	The error code is contained in the DDB (D.ERR(A2))
1	The specified PPN was not found in the MFD

The Z-flag will be set on a successful completion, and reset if an error occurs. If a -1 error occurs, the N-flag is also set.

All registers except D0, D6, D7, A1, and A6 are preserved.

Error message output is under the control of the D$BYP bit in the DDB flags byte.

$FNPPNX

Find a PPN in the MFD

AMOS 2.X or 1.X

The $FNPPNX routine has been superseded by the DIRSCH monitor call. $FNPPNX does not properly handle extended format directories and is provided only for compatibility with older software. Use of $FNPPNX in new software is discouraged.

The $FNPPNX routine provides a standard method of locating PPN entries within the MFD under either AMOS 2.X or 1.X. At run time, it checks the operating system version and calls the proper version of $FNPPN.

To find a PPN, you simply specify the PPN to be located, the DDB and buffer to be used, and the call $FNPPNX.

Before calling $FNPPNX, the following registers must be set up:

D1	Contains the PPN to be located in the low-order half of the register.
A2	Points to an INITed DDB to be used for the find operation.

$FNPPNX returns the following:

D0 Contains the completion code:

0	PPN was located successfully
-1	The error code is contained in the DDB (D.ERR(A2))
-2	The MFD is damaged in some way (AMOS 1.X only)
1	The specified PPN was not found in the MFD

A1 For AMOS 1.X only, if the PPN was found (D0 = 0), then A1 points to the PPN entry within the DDB buffer. If the PPN was not found (D0 = 1), then A1 points to the point at which the PPN would be inserted.

The Z-flag will be set on a successful completion, and reset if an error occurs. If a -1 or -2 error occurs, the N-flag is also set.

D.REC of the specified DDB is left containing the block number of the MFD block that contains the PPN being located. All registers except D0, D6, D7, A1, and A6 are preserved.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

Find User Description in USER.SYS

The $FNUSR routine provides a standard method of locating fields of the user description within the file DSK0:USER.SYS[1,2]. To find a user description, specify the user name to be located, the DDB and buffer to be used to open USER.SYS, and the call $FNUSR.

Before calling $FNUSR, the following registers must be set up:

A1	Points to an INITed DDB to be used to open USER.SYS.
A2	Points to a string specifying the user name you want to search for.

$FNUSR returns the following:

D0 Contains the completion code:

0	User name was located successfully
-1	The error code is contained in the DDB (D.ERR(A2))
1	The specified user name was not found.

D6	Relative block number of user name entry in USER.SYS.
A2	Indexes the user name description string in the buffer. Use US.xxx symbols to access the information (e.g., US.CLS for user class). The US.xxx symbols are documented in Appendix H, "User Description Symbols."

The Z-flag will be set on a successful completion, and reset if an error occurs. If a -1 error occurs, the N-flag is also set.

All registers except D0, A2, A6, D6, and D7 are preserved.

Error message output (from DDB type error codes) is under the control of the D$BYP bit in the DDB flags byte.

$GTARG

Get a Floating or String Argument from AlphaBASIC

The $GTARG routine provides a convenient way to convert AlphaBASIC XCALL subroutine arguments from binary, string, or floating point to binary. $GTARG will correctly handle one-, two-, three-, and four-byte binary variables (MAP type B), one-, two-, and four-byte integers (MAP type I), IEEE single- and double-precision variables (MAP types F,4 and F,8), AMOS floating point variables (MAP type F,6), and string variables holding string representations of positive integers. Floating point variables will be truncated toward minus infinity if necessary.

When using $GTARG, you simply specify the offset to the argument you wish converted. $GTARG then returns a 32-bit binary number.

Before calling $GTARG, set up the following registers:

D1	Contains the offset within the argument list for the argument you wish to convert (i.e., the first argument is at an offset of 2, the second is at an offset of 14, etc.).
A0	Contains the AlphaBASIC impure area pointer. This register is already set up when an XCALL routine is called by BASIC.
A3	Contains the argument list pointer that BASIC passes into the XCALL subroutine.
A5	Contains the arithmetic stack pointer that BASIC passes into the XCALL subroutine.

$GTARG returns the argument value as a 32-bit binary number in D1. If the argument you specified in D1 is out of range (e.g., you specified argument 5 when only 3 arguments were passed to the XCALL routine), the Z-bit will be returned and reset. On a successful call, the Z-bit will be set. You can therefore do a BNE to detect an error after a CALL $GTARG.

For further information on AlphaBASIC XCALL subroutines, see the *AlphaBASIC XCALL Subroutine User's Manual*.

$HSHFL

Generate a Standard File Hash

The $HSHFL routine scans a disk file and generates a standard hash total for the file. This is the same hash total as generated by the MAP and DIR/H programs.

In addition to returning a text representation of the hash total, $HSHFL also returns a 32-bit binary representation of the hash total, making comparison easier.

Before calling $HSHFL, the following registers must be set up:

D6	Contains binary flags describing what to do with the text representation of the hash total. If D6 is zero, no text representation will be output. If it is set to OT$TRM the result will be displayed on the terminal. If D6 contains OT$MEM the text representation will be placed in the memory buffer indexed by A6.
A2	Indexes a DDB describing the file to be hashed. The DDB must already be INITed.
A6	If D6 contains OT$MEM, A6 must index the memory buffer where the text representation of hash total will be placed.

$HSHFL returns with the following:

D0 Contains the completion code:

0	File was hashed successfully
-1	The error code is contained in the DDB (D.ERR(A2))
-2	The MFD is damaged in some way
1	The specified file could not be found
2	An invalid flag was specified in D6

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

$IDTIM

Input Date and Time

AMOS 2.X only

The $IDTIM subroutine accepts and packs a time or date, or both. Prior to calling $IDTIM, set up the registers as follows:

A2	Points to the string that is to be converted to packed date and time. Typically, A2 has been set up by a prior KBD monitor call.
D5	Contains flags, as follows:

Bit 0	Do not scan input for date.
Bit 1	Do not scan input for time.
Bit 2	Force colon as time field separator; otherwise, use separator defined in job's active Language Definition File.

Upon return from $IDTIM, A2 points past the time and date string, or points at an invalid character. D3 contains the date in separated format, and D4 contains the time in separated format.

If an error occurs while trying to parse the input, $IDTIM returns with the N-flag set on. (The $IDTIM call should always be followed by a BNE to an error routine.)

If the input specifies both time and date, the time must follow the date. Extraneous leading spaces are ignored. The time is considered to be one field—it is legal to omit the separator between the time field and the symbol that follows it, if any. Case differences in letters are ignored. Both "19nn" and "nn" imply the year "19nn." The input scan is completed by "AM," "PM," carriage-return, or a null.

Dates must be input with slash separators. Valid dates are from January 1, 1900 through December 31, 2155.

The system date can be set only to dates in the range January 1, 1980 through December 31, 2079. See Chapter 10 for further details

The date will be accepted in the order specified by the currently selected language, as defined in the language definition file for that language.

$IDTIMX

Input Date and Time

AMOS 2.X or 1.X

The $IDTIMX subroutine accepts and packs a time or date, or both. It has these advantages over $IDTIM: it is language-aware, it accepts more input formats, and it handles two-digit years more correctly.

Before calling $IDTIMX, set up the registers as follows:

A2	Points to the string that is to be converted to packed date and time. Typically, A2 has been set up by a prior KBD monitor call.
D5	Contains flags, as follows:

Bit 0	Do not scan input for date.
Bit 1	Do not scan input for time.
Bit 2	Force colon as time field separator; otherwise, use separator defined in job's active Language Definition File.

Upon return from $IDTIMX, A2 points past the time and date string, or points at an invalid character. D3 contains the date in separated format and D4 contains the time in separated format.

If an error occurs while trying to parse the input, $IDTIMX returns with the N-flag set on. (The $IDTIMX call should always be followed by a BNE to an error routine.)

If the input specifies both time and date, the time must follow the date. Extraneous leading spaces are ignored. The time separator must be the one specified in the current language definition file. The time is considered to be one field—it is legal to omit the separator between the time field and the symbol that follows it, if any. Case differences in letters are ignored. The input scan is completed by "AM," "PM," carriage-return, or a null.

Dates can use dashes, slashes, spaces, commas, or the character defined in the current language definition file as the day/month/year separator. The date will be accepted in the order specified by the currently selected language, as defined in the language definition file for that language.

Only dates between January 1, 1900 and December 31, 2155 are valid. A two-digit year from 80 through 99 will be interpreted as 1980 - 1999; a two-digit year of less than 80 will be interpreted as 2000 - 2079.

The system date can be set only to dates in the range January 1, 1980 through December 31, 2079. See Chapter 10 for further details

$INMFD

Initialize the MFD and Bitmap

AMOS 2.X only

The $INMFD routine provides a standard method of initializing the MFD and bitmap of a disk. To initialize the disk, simply specify a DDB referencing the disk to be initialized, and then call $INMFD. You must be logged into [1,2] to use this subroutine.

$INMFD works only with AMOS 2.X. If you want to support both AMOS 2.X and AMOS 1.X, use $INMFDX instead.

Before calling $INMFD, the following registers must be set up:

A2	Points to an INITed DDB to be used for the add operation.

$INMFD returns with the following:

D0	Contains the completion code:

0	Disk was initialized successfully
-1	The error code is contained in the DDB (D.ERR(A2))
1	You are not logged into [1,2]

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

All registers except D0, D6, D7, and A6 are preserved. This routine sets the D$ERC bit in the DDB flags byte.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$INMFDX

Initialize the MFD and Bitmap

AMOS 2.X or 1.X

The $INMFDX routine provides a standard method of initializing the MFD and bitmap of a disk under either AMOS 2.X or 1.X. At run time, it checks the operating system version and calls the proper version of $INMFD.

To initialize the disk, simply specify a DDB referencing the disk to be initialized, and then call $INMFDX. You must be logged into [1,2] to use this subroutine.

Before calling $INMFDX, the following registers must be set up:

A2	Points to an INITed DDB to be used for the add operation.

$INMFDX returns with the following:

D0 Contains the completion code:

0	Disk was initialized successfully
-1	The error code is contained in the DDB (D.ERR(A2))
1	You are not logged into [1,2]

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

All registers except D0, D6, D7, and A6 are preserved. This routine sets the D$ERC bit in the DDB flags byte.

Error message output (whether from DDB type error codes or from the "Damaged MFD" error) is under the control of the D$BYP bit in the DDB flags byte.

$KILPF

Kill a File Entry in Spooler Queue (Task Manager Spooler Only)

$KILPF stops the printing of a file in the printer queue, and/or removes the file from the queue. Only works with the Task Manager controlled printer spooler. Set up registers:

A4	Pointer to the DDB of the file to kill
D0	Printer name in RAD50 (1 LWORD)
D1	Sequence number (if present, A4 and D0 ignored)
D2	CPU number of spooler node (0 if default)

The following will be returned:

D2	Number of file entries killed
D1	If N-flag is set on return, this register will contain the ITC error code that occurred. If a non-ITC error code occurred, it will be returned in D0.
D0	If the N-flag is set on return, and D1 contains zero, a non-ITC error occurred. This register contains flags defining the error:

10000	aborted

If a file-oriented error occurs, the error code will be stored in D.ERR(A4).

$MSGLOG

Send a Message to the System Logger

Sends a message to the System Logger. Before calling $MSGLOG, set up the registers as follows:

D3	User code 0-999
A1	Index to user message string (up to 45 characters), null terminated.

After calling $MSGLOG, the Z flag returns 0 (is not set) if an argument error occurred, and returns 1 (is set) if the message was successfully sent. An argument error occurs if the user code is larger than 999 or if the index is zero.

$NETED

Convert a CPU Specification to an Ersatz Name

Accepts as an argument a CPU number, and returns a string containing the ersatz device name by which that CPU is known (if any). Before calling $NETED, set up the registers as follows:

A2	Points to a text buffer used to store the ersatz name
D1	Contains the CPU number whose ersatz name you wish to locate

After calling $NETED, the following will be returned:

A2	Updated pointed into the text buffer.

If $NETED was able to locate an ersatz name for the requested CPU number, the Z-bit will be set upon return. If no ersatz name was located, the Z-bit will be reset to zero. The text buffer pointed to by A2 upon calling $NETED must be a minimum of eight bytes long.

$ODTIM
Output Date and Time

The $ODTIM routine provides a standard way of outputting dates and times in a variety of formats. Prior to calling $ODTIM, set up the registers as follows:

A2	The destination pointer. If zero, then send output to the user's terminal. If non-zero and D5 bit 15 is zero, then output to the memory buffer indexed by A2. If non-zero and D5 bit 15 is one, then output to the DDB indexed by A2.
D3	Contains the date to be output, in separated format. If D3 contains a zero, then the current system date and time are used, and the contents of D4 are ignored.
D4	Contains the time to be output, in separated format. If D3 contains a zero, this register is ignored.
D5	Contains date and/or time formatting flags and other options as follows:

DESTINATION OPTIONS

Bit 15	If set and A2 is non-zero, treat A2 as a pointer to a DDB and do all output through that DDB. If clear, output to the user's terminal if A2 is zero, otherwise output to memory indexed by A2 and update A2 to point past the last character of the string on exit.

TIME AND DATE ORDERING AND FORMATTING

Bit 0	Omit date from the output, and ignore all other date formatting flags.
Bit 9	Omit time from the output, and ignore all other time formatting flags.
Bit 14	Affects numerical output for day, month and year only. If on, leading zeros are suppressed on all elements. This produces output suitable for use in an isolated message. If off, leading zeros are added if appropriate, producing output of a constant width that is useful for printing tables. Note that full month and weekday texts are never columniated, and time formats always have leading zeros. Overridden by Bit 18. See also Bit 19.
Bit 26	Output the time before the date, else the time will follow the date.

DATE FORMATTING

Date Punctuation

Bits 7-8	Date punctuation control (can be overridden by Bit 21):
Both off:	Use dashes, as 20-Feb-82
7 only:	Use spaces, as in 20 Feb 82; if Bit 6 is on and Bit 3 is off, add a comma, as in February 20, 1990
8 only:	Use slashes, as 2/20/82.
Both on:	Use the character defined as LD.DTS in the job's language definition file.
Bit 20	Do not output any commas in the date string. Overrides Bit 2 and Bit 6.
Bit 21	Do not output date punctuation. Overrides Bits 7 and 8.

Date Ordering

Bit 6	Output the day of the month after the month, otherwise output it before.

	Ignored if Bits 7 or 8 are set. A comma is appended if the day is immediately followed by the year and the month is not output as a number.
Bit 16	If set, the year precedes the day and month (which are printed in the order specified by Bit 6).

Date Items Output

Bit 1	Output the day of the week.
Bit 23	Do not output the day. Overrides Bit 6 and Bit 19.
Bit 24	Do not output the month. Overrides Bit 3 and Bit 4.
Bit 25	Do not output the year. Overrides Bit 5 and Bit 16.

Weekday Options

Bit 2	Use the full text for the weekday, otherwise use a three-letter abbreviation. Bit 22 can force the weekday to be output as a number in the range 0-6, overriding this bit. A comma will be added if alphabetic output is generated and the date follows the weekday. Requires Bit 1.
Bit 22	Output the weekday as a number (0-6) as the first element of date output. Overrides Bit 2, requires Bit 1 to be set.

Day Options

Bit 19	Add English day suffix (st, nd, rd, th) to the numerical day output. Forces Bit 14 on for day output.

Month Options

Bit 3	Output the month as a number 1-12; ignore Bit 4.
Bit 4	Use the full text for the month, otherwise use a three-letter abbreviation.

Year Options

Bit 5	Output a four-digit year, otherwise use a two-digit year. Bit 14 may disable leading zeros for two-digit years; Bit 18 may force them on.
Bit 18	Force leading zeros in two-digit years for years in the range xx00-xx09. Overrides Bit 14.

TIME FORMATTING

Bit 11	Use 12-hour and AM/PM, otherwise use 24-hour time.
Bit 28	If 24-hour clock, output "hrs" after the time string. Otherwise ignored.

Hours Options

Bit 12	Do not output a separator between the hour and the minute values.
Bit 27	Affects hours output: if 12-hour clock, suppress any leading zero. If 24-hour clock, ignored.

Seconds Options

Bit 10	Omit seconds; otherwise include seconds preceded by the time separator.

Punctuation Options

Bit 13	Force a colon as the time separator, otherwise use the character defined as LD.TMS in the job's language definition file.
Bit 17	Do not output separator between the minute and the second.

Notes

A zero in D5 produces columniated output of the form "03-Apr-81 01:06:03" on the user's terminal.

The default value for register D5 is -1. If D5 = -1, the date will be output in "normal" format. The normal format for American English is, for example: "Tuesday, March 24, 1981 01:12:54 PM." Other languages will use the specific format for the date as defined in the Language Definition File for that language.

Examples

Assume a language definition file containing the following:

Date Format = DMY	Time Format = 24-hour
Date Separator = Backslash	Time Separator = Period

On February 1, 2000, at 1:20 PM, $ODTIM returns the following:

Flag	Output	Notes
^H200	01-Feb-00	No date format bits set
^H288	01 02 00	Use spaces
^H308	01/02/00	Use slashes
^H2A8	01 02 2000	Use four-digit year
^H388	01\02\00	Use .LDF separator
^H4388	1\2\0	Suppress leading zeros
^H43A8	1\2\2000	Use four-digit year
^H200388	010200	Ignore date punctuation
^H2003A8	01022000	Same, four-digit year
^H210228	20000102	Year first, then day, month
^H210268	20000201	Year, month, day
^H10268	2000-02-01	Year, month, day: date punctuation
^H28A	Tue 01 02 00	Use weekday abbreviation
^H10262	Tue 2000-Feb-01	Year first, month first, punctuation
^H28E	Tuesday, 01 02 00	Full weekday with comma
^H2B6	Tuesday, 01 February 2000	
^H42B6	Tuesday, 1 February 2000	Ignore leading zeros
^H0C02B6	Tuesday, 1st February 2000	Ignore leading zeros, add day suffix
^H1C02B6	Tuesday 1st February 2000	Ignore all date commas
^H0C02F6	Tuesday, February 1st, 2000	Day after month, adds comma
^H42F6	Tuesday, February 1, 2000	No day suffix
^H1	13.20.00	24-hour clock
^H401	13.20	Ignore seconds
^H1C01	0120 PM	Add 12-hr clock, no hour/minute separator
^H801	01.20.00 PM	Add separators, seconds
^H2C01	01:20 PM	No seconds, force colon
^H21801	012000 PM	Add seconds, no separators
^H21001	132000	24-hr clock
^H0	01-Feb-00 13.20.00	Note .LDF time separators
^H0FFFFFFFF (-1)	Tuesday, 1 February 2000 13:20:00	Colons are forced

$ODTM2

Output Localized Date and Time

The $ODTM2 routine provides a standard way to output dates and times in the positional format described in the language definition (.LDF) file, while letting the user control the content and format of the output. Prior to calling $ODTM2, set up the registers as follows:

D3	Contains the date to be output in separated format. If D3 contains a zero, then the current system date and time are used, and the contents of D4 are ignored.
D4	Contains the time to be output in separated format. If D3 contains a zero, this register is ignored.
D5	Contains formatting flags. The flags are the same as those listed for $ODTIM, but $ODTIM2 forces the correct flags to be set according to the job’s active langauge definition file.
A2	The destination pointer. If zero, then send output the user's terminal. If non-zero and D5 bit 15 is zero, then output to the memory buffer indexed by A2. If non-zero and D5 bit 15 is one, then output to the DDB indexed by A2.

If the language definition file is available, $ODTM2 sets bits 6, 11, and 16, then calls the $ODTIM library routine.

Upon return, if A2 was used as a memory buffer pointer, it is updated.

A zero in D5 produces columniated output of the form "03-Apr-81 01:06:03" on the user's terminal.

The default value for register D5 is -1. If D5 = -1, the date will be output in "normal" format. The normal format for American English is, for example: "Tuesday, March 24, 1981 01:12:54 PM." Other languages will use the specific format for the date as defined in the Language Definition File for that language.

Here is an example use:

```
EXTERN    $ODTM2

CLR       D3
CLR       D5
MOV       #0,A2
CALL      $ODTM2
EXIT
```

This table shows the output from this routine for various date and time formats in the language definition file:

LDF Format	Output
M-D-Y 12hr	Jul-16-97 01:17:20 PM
D-M-Y 12hr	16-Jul-97 01:17:20 PM
Y-M-D 12hr	97-Jul-16 01:17:20 PM
D-M-Y 24hr	16-Jul-97 13:17:20

$OTCON
Output Connect Time

The $OTCON routine provides a way of unpacking and displaying connect time. This packed format is used within each JCB to store the connect time, and within the system communication area to store the up time.

Before calling $OTCON, the following registers must be set up:

D1	Contains the packed connect time to be displayed.
A2	If A2 is zero, the output is displayed on the user's terminal. If A2 is non-zero, it is used as a pointer to a memory buffer where the output is stored.

If A2 is non-zero, it is updated to point to the byte immediately following the last byte output by $OTCON. $OTCON preserves all other registers (except D6, D7, and A6).

$OTCPU
Output CPU Time

The $OTCPU routine provides a way of unpacking and displaying CPU time. This packed format is used within each JCB to store the CPU time used by the job.

$OTCPU will display as the decimal point the character defined as the decimal point in the current job's language definition file.

Before calling $OTCPU, the following registers must be set up:

D1	Contains the packed CPU time to be displayed.
A2	If A2 is zero, the output is displayed on the user's terminal. If A2 is non-zero, it is used as a pointer to a memory buffer where the output is stored.

If A2 is non-zero, it is updated to point to the byte immediately following the last byte output by $OTCPU. $OTCPU preserves all other registers (except D6, D7, and A6).

$PAKDT

Pack the Current Time and Date into Directory Format

The $PAKDT routine converts the current time and date into the internal 32-bit directory format. This format is used within the AMOS file structure for storing times and dates.

This format will represent dates in the range March 1, 1900 through December 31, 2027, inclusive.

$PAKDT does not require any setup prior to calling.

Upon completion $PAKDT returns with the following:

D0	Current date and time in packed directory format.

All registers except D0, D6, D7, and A6 are preserved.

Note that the internal directory format is packed in such a way that a simple 32-bit compare can be done to determine the relative position in time of the packed date and time.

The packed directory format may be unpacked via the $UNPDT subroutine.

$PAMFD

Pre-Allocate Permanent MFD Blocks

AMOS 2.X only

Under versions of AMOS prior to 2.0, this routine pre-allocated additional MFD blocks. This feature is no longer available and the $PAMFD routine has been converted to simply return to the caller. It is provided for compatibility with some older software which explicitly referenced it.

$PAMFD works only with AMOS 2.X. If you want to support both AMOS 2.X and AMOS 1.X, use $PAMFDX instead.

$PAMFDX

Pre-Allocate Permanent MFD Blocks

AMOS 2.X or 1.X

At run time, $PAMFD checks the operating system version and calls the proper version of $PAMFD. Under versions of AMOS prior to 2.0, this routine pre-allocated additional MFD blocks. This feature is no longer available and the $PAMFD routine has been converted to simply return to the caller. It is provided for compatibility with some older software which explicitly referenced it.

$SEND

Send a Text Message to Another Job

The $SEND routine allows you to send a text message to another job's terminal, located either on the same system or on a different system connected via AlphaNET. This subroutine is used by the SEND program to perform its functions.

Before calling $SEND, the following registers must be set up:

D2	Contains the name of the sending job, packed in RAD50. If this field is zero the current job's name will be used.
D3	Contains the CPU number of the sending job. If this field is zero the current job's CPU will be used.
D4	Contains the name of the job to send to, packed RAD50.
D5	Contains the CPU number of the job to send to. A zero in this register implies the current CPU.
A2	Contains a pointer to the ASCII text string to be sent to the other job. This string must be null terminated.

$SEND returns with the following:

D0	Contains the completion code:

1	The destination job could not be located
2	The destination job is guarded against messages
3	The destination job is busy
4	The destination job does not have a terminal attached to it
5	The remote CPU is not responding
6	An unrecognized error code was returned by the destination system
7	An unexpected error was returned by the inter-task communication system

In addition to the above codes, the Z-flag will be set on a successful completion, and reset if an error occurs.

$SPLFL

Queue Up a File to the Spooler

Works for both the memory resident spooler and the Task Manager print spooler programs. The routine will first try the memory resident spooler, and then the Task Manager spooler. Set up these registers:

A4	Pointer to the DDB of the file
A2	Pointer to the data (24. words)

	PRINTR	4 bytes	Printer name in RAD50 (0 if default)
	CPU	4 bytes	CPU number of spooler node (0 if default)
	PSW	4 bytes	Positive switches (See "Switch Settings," below)
	NSW	4 bytes	Negative switches (See "Switch Settings," below)
	COPIES	2 bytes	Number of copies
	BANNER	4 bytes	Pointer to BANNER text buffer (max. 50 bytes)
	LPP	2 bytes	Lines per page
	WIDTH	2 bytes	Width of text lines
	FORMS	4 bytes	Type of form (RAD50)
*	PRI	2 bytes	Priority # of the file
*	AFTERD	4 bytes	Run date (separated format)
*	AFTERT	4 bytes	Run time
*	RSTART	2 bytes	/RESTART{:n}
*	START	2 bytes	/START:n
*	FINISH	2 bytes	/FINISH:n
*	LIMIT	2 bytes	/LIMIT{:n}

* = Task Manager Spooler only

The following is returned:

D2 The sequence number of the file in the queue (if using the Task Manager spooler)
If D2 = 0 and there is no error, the memory resident spooler is installed

If an error occurs, the N-flag will be set, and the following registers will specify which error occurred:

D1 ITC error code

D0 Other error codes:

Code	Meaning
1	Printer not found.
2	Printer already turned off.
4	Printer already turned on.
10	Queue entry specified by //SEQ not found.
20	Printing of the file has already started.
40	Queue file is full; not enough free queue records for request.
100	Spooler file error: including queue file error.
200	Print file error.
2000	Illegal switch(es)
4000	Not enough queue blocks for spooler request (memory resident spooler).
10000	Aborted.

If a file oriented error occurs, the error code will be in D.ERR(A4).

SWITCH SETTINGS

You may select/deselect spooler options by placing the appropriate values into PSW (Positive Switch) to select the option and NSW (Negative Switch) to deselect the option as defined below. The bit values shown below are in octal.

You should use the enhanced ^C abort flag if the calling program does not run in line mode and ^C interrupts are enabled. If the flag is not set, and the program is in image or data mode, a user cannot abort spooling by pressing CTRL/C. If the flag is set and the user presses CTRL/C, $SPLFL exits with D0 set to 10000, and any pending keyboard input is discarded.

PSW - Activate Option

Value		Option
1		/COPIES:n
2		/DELETE
4		/BANNER{:text}
10		/HEADER
20		/FF
40		/LPP:n
100		/WIDTH:n
200		/WAIT
400		/FORMS:x
4000	*	/SUSPEND
10000	*	/REVIVE
20000	*	/PRIORITY:n
40000	*	/AFTER:{+}mm-dd-yy {@hh:mm AM/PM}
100000	*	/INFORM
200000	*	/RESTART{:n}
400000	*	/START:n
1000000	*	/FINISH:n
2000000	*	/LIMIT{:n}
20000000		Enhanced ^C abort handling

NSW - Inactivate Option

Value		Option
2		/DELETE
4		/BANNER{:text}
10		/HEADER
20		/FF
200		/WAIT
4000	*	/SUSPEND
10000	*	/REVIVE
40000	*	/AFTER:{+}mm-dd-yy {@hh:mm AM/PM}
100000	*	/INFORM
200000	*	/RESTART{:n}
400000	*	/START:n
1000000	*	/FINISH:n
2000000	*	/LIMIT{:n}

* = Task Manager Spooler only (in both tables)

$STFRM

Set Forms Type on a Printer (Task Manager Spooler Only)

$STFRM allows you to set the type of printer form needed for the printing of a file. It works only with the Task Manager spooler. Set up the registers:

D0	Printer name in RAD50 (1 LWORD)
D1	Form name in RAD50

If an error occurs, the N-flag will be set, and the following registers will specify which error occurred.

D1	ITC error code
D0	Other error code:

400	printer name not specified
1000	not logged in to OPR:
10000	aborted

$SYSID

Get a Symbolic System Name for a Network CPU

$SYSID takes as an argument a CPU number and returns the symbolic name that the remote CPU calls itself. Before calling $SYSID, set up the following registers:

A3	Points to a buffer to be used for sending and receiving Inter Task Communication (ITC) messages. This buffer must be at least 512 bytes in length.
D1	Contains the CPU number for which you wish to receive the system ID.

After returning from $SYSID, the following registers will be returned:

D1	Completion code.
A2	Points to a null terminated string containing the system ID.
A3	Points to a null terminated string containing the system name and version.

Upon return, the Z-bit will be set is $SYSID was successful, and will be reset if the remote system was not available (not responding).

In addition, the N-bit will be set if $SYSID received a fatal ITC error code while attempting to communicate with the remote system. In this case, the ITC error code will be stored in D1.

$UNPDT

Unpack Directory Format Date and Time into Separated Format

The $UNPDT routine converts a packed date and time in the internal 32-bit directory format, into a date and a time in separated format. The packed directory format is used within the AMOS file structure for storing times and dates.

This format will represent dates in the range March 1, 1900 through December 31, 2027, inclusive.

Before calling $UNPDT set up the following:

D0	Date and time to be unpacked from directory format.

Upon completion $UNPDT returns with the following:

D3	Date in separated format.
D4	Time in separated format.

All registers except D0, D3, D4, D6, D7, and A6 are preserved.

Note that the internal directory format is packed in such a way that a simple 32-bit compare can be done to determine the relative position in time of the packed date and time.

The current date and time can be packed into directory format via the $PAKDT subroutine.

$UPDSW

Update Switches in a Spooler Entry
(Task Manager Spooler Only)

Allows you to change switch(es) for a file that is already in the printer spooler. Only works with the Task Manager controlled printer spooler. Set up the registers:

A2 Pointer to switch-related data (20. words):

PSW	4 bytes	Positive switches
NSW	4 bytes	Switches
COPIES	2 bytes	Number of copies
BANNER	4 bytes	Pointer to BANNER text buffer (max. 50 bytes, null terminated)
LPP	2 bytes	Lines per page
WIDTH	2 bytes	Width of text lines
FORMS	4 bytes	Name of form (RAD50)
PRI	2 bytes	Priority # of the file
AFTERD	4 bytes	Run date (separated format)
AFTERT	4 bytes	Run time
RSTART	2 bytes	/RESTART{:n}
START	2 bytes	/START:n
FINISH	2 bytes	/FINISH:n
LIMIT	2 bytes	/LIMIT{:n}

D0 Printer name in RAD50 (1 LWORD. 0 if default)

D1 Sequence number of the queue entry to update (1 LWORD)

D2 CPU number of the spooler node (0 if default)

If an error occurs, the N-flag will be set, and the following registers will specify which error occurred.

D1 ITC error code

D0 Other error code:

2000	illegal switch(es)
10000	aborted
20000	illegal sequence number

$YESNO

Accept and Decipher "YES/NO" Responses in Any Language

$YESNO allows you to check any yes/no input from a user. This subroutine accepts input from the user which is compared against the YES and NO responses defined in the currently selected language definition table. This allows a program to be "language independent" without having to concern itself with the implementation details of the language definition files.

The KBD (for line mode input) and the TIN (image or data mode input) monitor calls are performed in this routine for you. If a Control-C was used while waiting for input, all flags will be cleared and $YESNO will return back to your program. Therefore, a ^C check should be done before anything else.

If you wish image mode input, your program must put the job into image mode before calling $YESNO, otherwise, you need no preparation for the call.

$YESNO returns:

Z-flag	Set for affirmative response. Tested for by a BEQ.
N-flag	Set for negative response. Tested for by a BMI.
C-flag	Set for a Carriage return response (i.e., default response). Tested for by a BCS.
V-flag	Set for a bad response (anything else). Tested for by a BVS.

If the job running $YESNO entered the routine in image mode, that flag will be cleared upon return. All registers are preserved except A6, D6, and D7.

Appendix E
Alphabetic Listing of AMOS Monitor Calls

The following is a quick reference to all AMOS monitor calls:

ALF	tests the character indexed by A2 for alphabetic
AMOS	executes AMOS command without exiting current program
ASSIGN	assigns a non-sharable device to a job
BACKSP	perform backspace operation on tape
BUFSTS	inquire on buffer status for printer device
BYP	bypasses all spaces and tabs in the string indexed by A2
CHGMEM	changes the size of a user memory module
CHKMSG	check for received messages
CHPROT	change file protection
CLOSE	close a logical dataset
CLOSEK	close a logical dataset, but keep it locked for the user who opened it.
CLSMSG	close a message socket
COMINT	select communications interrupt service routines
COMRST	read communications port status flags
COMSET	set communications port parameters
COMWST	write communications port status flags
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	de-assigns a non-sharable device from a job
DELMEM	deletes a user memory module from his partition
DELSHM	deletes a block of shared memory
DEVCHR	get device characteristics
DIRACC	request and set up access to directory
DIRALC	allocate a new directory item
DIRDEL	delete a directory item
DIRREP	replace an existing directory item
DIRSCH	search for specified item in directory
DQTIMR	remove an active timer queue entry
DSKALC	allocates next available record on disk and returns block number
DSKCTG	allocates a contiguous file for random processing
DSKDEA	deallocates a record on disk and makes it available for use again
DSKDEL	deletes a file from a file-structured device
DSKDRL	sets re-entrant directory lock for a specific user's directory
DSKDRU	clears re-entrant directory lock for a specific user's directory
DSKFRE	get number of free blocks on device
DSKINI	initialize a disk

DSKMNT	mounts a file structured device
DSKREN	renames a file on a file-structured device
DSKUMT	unmounts a file structured device
DVSTAT	get tape device status
ERASE	perform erase function on tape drive
ERRMSG	converts standard system error code to text
EXIT	exits from user program and returns to monitor command mode
FADD	performs floating point addition using Alpha Micro format
FATOID	convert Alpha Micro format floating point to double precision IEEE format
FATOIS	convert Alpha Micro format floating point to single precision IEEE format
FCMP	performs floating point comparison using Alpha Micro format
FCVT	converts an Alpha Micro format floating point number to text
FCVTD	converts a double precision IEEE format floating point number to text
FCVTS	converts a single precision IEEE format floating point number to text
FDIV	performs floating point division using Alpha Micro format
FETCH	fetches a module from disk into user memory unless already in memory
FFTOA	converts an Alpha Micro format floating point number to ASCII
FFTOAX	converts an Alpha Micro format floating point number to extended ASCII
FFTOL	converts an Alpha Micro format floating point number to a longword
FFTOX	converts an Alpha Micro format floating point number to 40-bit extended format
FILINB	input a byte from a sequential file
FILINL	input a longword from a sequential file
FILINW	input a word from a sequential file
FILNAM	processes a filename specification indexed by A2 into RAD50 format
FILOTB	output a byte to a sequential file
FILOTL	output a longword to a sequential file
FILOTW	output a word to a sequential file
FIDTOA	convert double precision IEEE format floating point to Alpha Micro format
FISTOA	convert single precision IEEE format floating point to Alpha Micro format
FLTOF	converts a longword to Alpha Micro floating point format
FMARK	find file mark on specified magnetic tape unit
FMARKR	read in reverse to find file mark on specified magnetic tape unit
FMUL	performs floating point multiplication using Alpha Micro format
FPWR	multiplies an Alpha Micro format floating point number by a power of ten
FSPEC	processes a complete file spec indexed by A2 and sets up DDB
FSUB	performs floating point subtraction using Alpha Micro format
FXTOF	converts 40-bit extended format to Alpha Micro floating point format
GDATEI	gets system date in internal format
GDATES	gets system date in separated format
GET	read a record from a file
GETL	read a record from a file and leave it locked
GETX	read a record from a file regardless of locking
GETMEM	allocates a user memory module in his partition
GETSHM	gets a block of shared memory
GTDEC	converts a decimal number indexed by A2 to binary, returns it in D1
GTFLFD	input a double precision IEEE format floating point number
GTFLFS	input a single precision IEEE format floating point number
GTFLT	input a floating point number in Alpha Micro format
GTFLTD	input a double precision IEEE format floating point number from a file
GTFLTF	input a floating point number in Alpha Micro format from a file

GTFLTS	input a single precision IEEE format floating point number from a file
GTIMEI	gets system time in internal format
GTIMES	gets system time is separated format
GTLANG	returns a pointer to the language definition table
GTOCT	converts an octal number indexed by A2 into binary and returns it in D1
GTPARM	read VCR interface parameters
GTPPN	converts a p,pn format indexed by A2 into binary and returns it in D1
ICOFF	turns off CPU instruction cache
ICON	turns on CPU instruction cache
INIT	initializes a dataset driver block (DDB) for I/O processing
INPUT	performs a logical record input I/O function on an open dataset
INPUTL	performs a logical record input I/O function to update a record
INPUTX	performs a logical record input I/O function regardless of locking
JCBIDX	index a job control block
JLOCK	prevents context switches and allows current user to run
JLOCKI	prevents context switches and allows current user to run while allowing current I/O to complete.
JOBIDX	set an index to a job control block item for the current job
JRUN	restores a waiting job to the run request state
JUNLOK	enables context switches (reverses effect of JLOCK)
JWAIT	sets your job into a wait state
KBD	accepts input from user terminal keyboard (character or line mode)
LCS	converts one character in D1 to lower case
LEVEL7	enters Level7 debugger
LIN	tests the character indexed by A2 for valid end-of-line character
LOCKF	relocks a file that has already been OPENed
LOKSHM	locking shared memory block
LOOKUP	looks for a specific file on disk and returns information about it
MDDIAL	dial phone number on modem
MDOFF	disable modem
MDON	enable modem
MDREQ	request a modem with specified characteristics
MDRTN	return a modem to available modem pool
MDSET	set modem parameters
NUM	tests the character indexed by A2 for numeric
OCVT	converts a binary value to octal and prints it on the user terminal
OFILE	outputs a file specification
OPENA	opens a logical dataset for appending
OPENI	opens a logical dataset for input
OPENO	opens a logical dataset for output
OPENR	opens a logical dataset for random access
OPNMSG	opens a message socket
OUT	generalized output byte call
OUTCR	generalized output string with carriage return call
OUTI	generalized output immediate call
OUTL	generalized output string call
OUTPTL	performs a logical record output I/O function for a new record
OUTPUT	performs a logical record output I/O function on an open dataset
OUTS	generalized output string call
OUTSP	generalized output string with space call

PACK	packs an ASCII triplet into its RAD50 code
PCALL	invokes program as subroutine
PFILE	prints a complete file specification on user terminal from a DDB
PHDR	macro that defines a program header area
PLOCK	lock a job and all related processes
PRNAM	prints a filename specification on user terminal from its packed format
PRPPN	prints a p,pn specification on user terminal from its packed format
PUNLOK	unlock a job and all related processes
PUT	write a record to a file
PUTL	create a record in a file
QADD	adds a queue block to the end of a queue list
QADDL	links a queue block to the end of a queue
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
QINSL	links a queue block into a queue at a defined point
QRET	removes a queue block from a queue list and returns it to the free list
QUNL	unlinks a queue block from a specified queue
RCVMSG	receive a message
READ	performs a physical record read I/O function on a dataset
RETNSN	re-tensions the tape on a specified streaming tape unit
REWIND	rewind magnetic tape on specified magnetic tape unit
RLSE	releases control of a semaphore and allows waiting job to access source
RQST	requests control of a semaphore to access source or wait in wait chain
RSTCON	resets (SP) to the base of the stack within current context
RTCRT	perform TCRT function on remote terminal
RTNMSG	return from message service system
SDATES	sets the system date from separated format
SETMSG	set MSQ/MSR status
SICLR	clear software interrupt request
SIDIS	disable software interrupts
SIMSK	select software interrupt enable mask
SIRTN	return from software interrupt
SISET	set software interrupt request
SISTS	get software interrupt enable status
SITIMR	request software interrupt after specified time interval
SIWAIT	wait for software interrupt
SKIPN	skip over tape blocks
SLEEP	puts the user job to sleep for a specified number of line clock ticks
SMSG	display system message
SNDMSG	send a message
SPAWN	spawns jobs to perform specific tasks
SRCH	searches for a named memory module and returns its address
STDERR	perform standard error processing
STIMES	sets system time from separated format
STPARM	set VCR interface parameters
SUPVR	places job in supervisor state
SVLOK	disables interrupts
SVUNLK	enables interrupts
SYNC	flushes write cache on demand
TAB	sends a tab character to the user terminal

TAPDEN	select tape drive density
TAPSPD	select tape drive speed
TAPTYP	select tape drive transport type
TAPST	read tape status of specified magnetic tape unit
TBUF	queues up a variable length data buffer for output to a terminal
TCBIDX	index terminal control block
TCKI	check terminal input buffer for input present
TCRT	executes the special function CRT routine in the active terminal driver
TDVCNG	change terminal drivers
TIMER	adds an entry to the system timer queue
TIN	reads one character from the user terminal input buffer
TINIT	initiate terminal output
TOUT	sends one character to the user terminal output buffer
TRM	tests the character indexed by A2 for a valid termination character
TRMBFQ	adds a data buffer to the active output queue of a terminal
TRMCHR	defines terminal characteristics for color terminals
TRMICP	processes one input character (used within terminal drivers)
TRMOCP	processes one output character (used within terminal drivers)
TRMRST	read terminal status flags
TRMWST	write terminal status flags
TTY	outputs one character to the user terminal
TTYI	outputs an in-line 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
UCS	converts one character in D1 to upper case
UNLKSHM	unlock a block of shared memory
UNLOAD	unloads a specified tape unit
UNLOKF	unlocks a file without closing it
UNLOKR	unlocks a record read by an INPUTL call, but not yet updated
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
VCVT	unpacks and display a version number
VSRCH	perform VCR interface search function
WAKE	wakes a job out of sleep state
WRITE	performs a physical record write I/O function on a dataset
WRTFM	write a file mark to specified magnetic tape unit
WTMSG	wait for receipt of a message

Appendix F
Character Sets

AMOS uses a single-byte character set. Such a character set can represent 256 different characters. The character set is aligned with a number of international standards.

A SHORT HISTORY OF CHARACTER SETS

The first international standard was set in 1965 by ECMA (European Computer Manufacturer's Association) and was known as ECMA-6. The character set was adopted by other standards bodies, and is also known as US-ASCII, DIN 66003, and ISO 646. The standard only defined a basic alphabet, and did not allow for national characters in use in many European countries. Such characters were incorporated by specifying twelve code points (see Note 1 in the table below) as being places where replacement characters could be defined. For example, Germany defined the letter Ä at code point 91, where the [character was located. These character sets were called the "national ISO 646 variants". Portability of files containing such characters were low.

In 1981, the IBM PC introduced an 8-bit character set with Code Page 437, a character set with many special characters. In 1982 DEC MCS (Multi Language Character Set) was released. This character set was very similar to ISO 6937/2, which in turn is almost identical to the modern standard for 8-bit character sets, ISO 8859. In 1985 ECMA standardized ECMA-94, which dealt with almost all European languages. ECMA-94 was taken up by ISO, as ISO 8859-1 through 8859-4, and standardized in 1987.

Microsoft released MS-DOS 3.3 in 1987, which used Code Page 850. This code page uses all the characters from ISO 8859-1, plus a few extra at code points representing the non-printing characters. A second code page, Code Page 819, is fully ISO 8859-1 compliant.

THE ISO 8859 FAMILY OF STANDARDS AND AMOS

The ISO 8859-x character sets are designed for maximum interoperability and portability. All of them are a superset of US-ASCII and will render English text properly. The code points 0xA0 through 0xFF are used to represent national characters, while the characters in the range 0x20 through 0x7F are the same as in the ISO 646 (US-ASCII) character set. Thus ASCII text is a subset of all ISO 8859 character sets, and will be rendered properly by them. The code points 0x80 through 0x9F are earmarked as extended control characters and are not used for encoding characters.

The ISO 8859 family of standards consists of:

8859-1	For Europe, Latin America, the Caribbean, Canada, and Africa
8859-2	For Eastern Europe
8859-3	For SE Europe, and a miscellany of alphabets, such as Esperanto, and Maltese
8859-4	For Scandinavia, and the Baltic states (mostly covered by 8859-1 also)
8859-5	For languages using the Cyrillic alphabet

8859-6	For languages using Arabic
8859-7	For modern Greek
8859-8	For Hebrew
8859-9	Known as Latin-5. The same as 8859-1 except for Turkish instead of Icelandic characters
8859-10	Known as Latin-6, for Lappish, Nordic, and Eskimo languages

ISO 8859-1 (also known as ISO Latin-1) has the required characters to display most Western European languages. It supports Afrikaans, Basque, Catalan, Danish, Dutch, English, Faeroese, Finnish, French, Galician, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish and Swedish. It cannot support Welsh, due to two missing characters (Latin Letter W with circumflex and Latin Letter Y with circumflex). It is the preferred encoding for the Internet.

AMOS follows this lead, and expects 8-bit aware software to use these ISO standards.

In passing, the ISO 8859-1 standard is a subset of the Unicode 1.x and 2.0 standards, which use 16-bit character sets to encode most of the world's alphabets. Unicode has aligned itself with a further ISO standard for 32-bit character sets, ISO 10646-1:1993. There are several mappings available (such as UTF-8) which can map Unicode characters to a variable length 8-bit based encoding.

Char-acter	Also Called	Octal Value	Decimal Value	Hex Value	ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name	Also Known As	Type.	See Note
NULL		0	0	0		Null	Cc	
SOH		1	1	1		Start of Heading	Cc	
STX		2	2	2		Start of Text	Cc	
ETX		3	3	3		End of Text	Cc	
EOT		4	4	4		End of Transmission	Cc	
ENQ		5	5	5		Enquiry	Cc	
ACK		6	6	6		Acknowledge	Cc	
BEL		7	7	7		Bell	Cc	
BS		10	8	8		Backspace	Cc	
HT		11	9	9		Character Tabulation (Tab)	Cc	
LF		12	10	A		Line Feed	Cc	
VT		13	11	B		Line Tabulation (Vertical Tab)	Cc	
FF		14	12	C		Form Feed	Cc	
CR		15	13	D		Carriage Return	Cc	
SO		16	14	E		Shift Out	Cc	
SI		17	15	F		Shift In	Cc	
DLE		20	16	10		Data Link Escape	Cc	
DC1		21	17	11		Device Control One	Cc	
DC2		22	18	12		Device Control Two	Cc	
DC3		23	19	13		Device Control Three	Cc	
DC4		24	20	14		Device Control Four	Cc	
NAK		25	21	15		Negative Acknowledge	Cc	
SYN		26	22	16		Synchronous Idle	Cc	
ETB		27	23	17		End of Transmission Block	Cc	
CAN		30	24	18		Cancel	Cc	
EM		31	25	19		End of Medium	Cc	
SUB		32	26	1A		Substitute	Cc	
ESC		33	27	1B		Escape	Cc	
FS	IS4	34	28	1C		File Separator	Cc	
GS	IS3	35	29	1D		Group Separator	Cc	
RS	IS2	36	30	1E		Record Separator	Cc	

Char-acter	Also Called	Octal Value	Decimal Value	Hex Value	ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name	Also Known As	Type.	See Note
US	IS1	37	31	1F		Unit Separator	Cc	
SP		40	32	20	Space		Zs	
!		41	33	21	Exclamation Mark		Po	
"		42	34	22	Quotation Mark		Po	
#		43	35	23	Number Sign	(Hash)	So	1
$		44	36	24	Dollar Sign		Sc	1
%		45	37	25	Percent Sign		Po	
&		46	38	26	Ampersand		So	
'		47	39	27	Apostrophe	Apostrophe-Quote	Po	
(		50	40	28	Left Parenthesis	Opening Parenthesis	Ps	
)		51	41	29	Right Parenthesis	Closing Parenthesis	Pe	
*		52	42	2A	Asterisk		So	
+		53	43	2B	Plus Sign		Sm	
,		54	44	2C	Comma		Po	
-		55	45	2D	Hyphen-Minus	Minus Sign	Pd	
.		56	46	2E	Full Stop	Period	Po	
/		57	47	2F	Solidus	Slash	Po	
0		60	48	30	Digit Zero		Nd	
1		61	49	31	Digit One		Nd	
2		62	50	32	Digit Two		Nd	
3		63	51	33	Digit Three		Nd	
4		64	52	34	Digit Four		Nd	
5		65	53	35	Digit Five		Nd	
6		66	54	36	Digit Six		Nd	
7		67	55	37	Digit Seven		Nd	
8		70	56	38	Digit Eight		Nd	
9		71	57	39	Digit Nine		Nd	
:		72	58	3A	Colon		Po	
;		73	59	3B	Semicolon		Po	
<		74	60	3C	Less-Than Sign		Sm	
=		75	61	3D	Equals Sign		Sm	
>		76	62	3E	Greater-Than Sign		Sm	
?		77	63	3F	Question Mark		Po	
@		100	64	40	Commercial At		Po	1
A		101	65	41	Latin Capital Letter A		Lu	
B		102	66	42	Latin Capital Letter B		Lu	
C		103	67	43	Latin Capital Letter C		Lu	
D		104	68	44	Latin Capital Letter D		Lu	
E		105	69	45	Latin Capital Letter E		Lu	
F		106	70	46	Latin Capital Letter F		Lu	
G		107	71	47	Latin Capital Letter G		Lu	
H		110	72	48	Latin Capital Letter H		Lu	
I		111	73	49	Latin Capital Letter I		Lu	
J		112	74	4A	Latin Capital Letter J		Lu	
K		113	75	4B	Latin Capital Letter K		Lu	
L		114	76	4C	Latin Capital Letter L		Lu	
M		115	77	4D	Latin Capital Letter M		Lu	
N		116	78	4E	Latin Capital Letter N		Lu	
O		117	79	4F	Latin Capital Letter O		Lu	
P		120	80	50	Latin Capital Letter P		Lu	
Q		121	81	51	Latin Capital Letter Q		Lu	
R		122	82	52	Latin Capital Letter R		Lu	
S		123	83	53	Latin Capital Letter S		Lu	
T		124	84	54	Latin Capital Letter T		Lu	
U		125	85	55	Latin Capital Letter U		Lu	
V		126	86	56	Latin Capital Letter V		Lu	

Char-acter	Also Called	Octal Value	Decimal Value	Hex Value	ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name	Also Known As	Type.	See Note
W		127	87	57	Latin Capital Letter W		Lu	
X		130	88	58	Latin Capital Letter X		Lu	
Y		131	89	59	Latin Capital Letter Y		Lu	
Z		132	90	5A	Latin Capital Letter Z		Lu	
[		133	91	5B	Left Square Bracket	Opening Square Bracket	Ps	1
\		134	92	5C	Reverse Solidus	Backslash	Po	1
]		135	93	5D	Right Square Bracket	Closing Square Bracket	Pe	1
^		136	94	5E	Circumflex Accent	Spacing Circumflex; Caret	Lm	1
_		137	95	5F	Low Line	Spacing Underscore; Underscore	So	
`		140	96	60	Grave Accent	Spacing Grave	Lm	1
a		141	97	61	Latin Small Letter A		Ll	
b		142	98	62	Latin Small Letter B		Ll	
c		143	99	63	Latin Small Letter C		Ll	
d		144	100	64	Latin Small Letter D		Ll	
e		145	101	65	Latin Small Letter E		Ll	
f		146	102	66	Latin Small Letter F		Ll	
g		147	103	67	Latin Small Letter G		Ll	
h		150	104	68	Latin Small Letter H		Ll	
i		151	105	69	Latin Small Letter I		Ll	
j		152	106	6A	Latin Small Letter J		Ll	
k		153	107	6B	Latin Small Letter K		Ll	
l		154	108	6C	Latin Small Letter L		Ll	
m		155	109	6D	Latin Small Letter M		Ll	
n		156	110	6E	Latin Small Letter N		Ll	
o		157	111	6F	Latin Small Letter O		Ll	
p		160	112	70	Latin Small Letter P		Ll	
q		161	113	71	Latin Small Letter Q		Ll	
r		162	114	72	Latin Small Letter R		Ll	
s		163	115	73	Latin Small Letter S		Ll	
t		164	116	74	Latin Small Letter T		Ll	
u		165	117	75	Latin Small Letter U		Ll	
v		166	118	76	Latin Small Letter V		Ll	
w		167	119	77	Latin Small Letter W		Ll	
x		170	120	78	Latin Small Letter X		Ll	
y		171	121	79	Latin Small Letter Y		Ll	
z		172	122	7A	Latin Small Letter Z		Ll	
{		173	123	7B	Left Curly Bracket	Opening Curly Bracket	Ps	1
|		174	124	7C	Vertical Line	Vertical Bar	So	1
}		175	125	7D	Right Curly Bracket	Closing Curly Bracket	Pe	1
~		176	126	7E	Tilde		So	1
DEL		177	127	7F		Delete	Cc	
PAD		200	128	80		Padding Character	Cc	
HOP		201	129	81		High Octet Preset	Cc	
BPH		202	130	82		Break Permitted Here	Cc	
NBH		203	131	83		No Break Here	Cc	
IND		204	132	84		Index	Cc	
NEL		205	133	85		Next Line	Cc	
SSA		206	134	86		Start of Selected Area	Cc	
ESA		207	135	87		End of Selected Area	Cc	
HTS		210	136	88		Character Tabulation Set	Cc	
HTJ		211	137	89		Character Tabulation with Justification	Cc	
VTS		212	138	8A		Line Tabulation Set	Cc	
PLD		213	139	8B		Partial Line Forward	Cc	
PLU		214	140	8C		Partial Line Backward	Cc	

Char-acter	Also Called	Octal Value	Decimal Value	Hex Value	ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name	Also Known As	Type.	See Note
RI		215	141	8D		Reverse Line Feed	Cc	
SS2		216	142	8E		Single-Shift Two	Cc	
SS3		217	143	8F		Single-Shift Three	Cc	
DCS		220	144	90		Device Control String	Cc	
PU1		221	145	91		Private Use One	Cc	
PU2		222	146	92		Private Use Two	Cc	
STS		223	147	93		Set Transmit State	Cc	
CCH		224	148	94		Cancel Character	Cc	
MW		225	149	95		Message Waiting	Cc	
SPA		226	150	96		Start of Guarded Area	Cc	
EPA		227	151	97		End of Guarded Area	Cc	
SOS		230	152	98		Start of String	Cc	
SGCI		231	153	99		Single Graphic Character Introducer	Cc	
SCI		232	154	9A		Single Character Introducer	Cc	
CSI		233	155	9B		Control Sequence Introducer	Cc	
ST		234	156	9C		String Terminator	Cc	
OSC		235	157	9D		Operating System Command	Cc	
PM		236	158	9E		Privacy Message	Cc	
APC		237	159	9F		Application Program Command	Cc	
NBSP		240	160	A0	No-Break Space		Zs	
¡		241	161	A1	Inverted Exclamation Mark		Po	
¢		242	162	A2	Cent Sign		Sc	
£		243	163	A3	Pound Sign		Sc	
¤		244	164	A4	Currency Sign		Sc	
¥		245	165	A5	Yen Sign		Sc	
¦		246	166	A6	Broken Bar		So	
§		247	167	A7	Section Sign		So	
¨		250	168	A8	Diaeresis		Lm	
©		251	169	A9	Copyright Sign		So	
ª		252	170	AA	Feminine Ordinal Indicator		So	
«		253	171	AB	Left-Pointing Double Angle Quotation Mark		Ps	
¬		254	172	AC	Not Sign		Sm	
-		255	173	AD	Soft Hyphen		Po	
®		256	174	AE	Registered Sign		So	
¯		257	175	AF	Macron		Lm	
°		260	176	B0	Degree Sign		So	
±		261	177	B1	Plus-Minus Sign		Sm	
²		262	178	B2	Superscript Two		So	
³		263	179	B3	Superscript Three		So	
´		264	180	B4	Acute Accent		Lm	
µ		265	181	B5	Micro Sign		So	
¶		266	182	B6	Pilcrow Sign		So	
·		267	183	B7	Middle Dot		Po	
¸		270	184	B8	Cedilla		Lm	
¹		271	185	B9	Superscript One		So	
º		272	186	BA	Masculine Ordinal Indicator		So	
»		273	187	BB	Right-Pointing Double Angle Quotation Mark		Pe	
¼		274	188	BC	Vulgar Fraction One Quarter		So	
½		275	189	BD	Vulgar Fraction One Half		So	
¾		276	190	BE	Vulgar Fraction Three		So	

Char-acter	Also Called	Octal Value	Decimal Value	Hex Value	ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name	Also Known As	Type.	See Note
					Quarters			
¿		277	191	BF	Inverted Question Mark		Po	
À		300	192	C0	Latin Capital Letter A With Grave		Lu	
Á		301	193	C1	Latin Capital Letter A With Acute		Lu	
Â		302	194	C2	Latin Capital Letter A With Circumflex		Lu	
Ã		303	195	C3	Latin Capital Letter A With Tilde		Lu	
Ä		304	196	C4	Latin Capital Letter A With Diaeresis		Lu	
Å		305	197	C5	Latin Capital Letter A With Ring Above		Lu	
Æ		306	198	C6	Latin Capital Ligature AE		Lu	2
Ç		307	199	C7	Latin Capital Letter C With Cedilla		Lu	
È		310	200	C8	Latin Capital Letter E With Grave		Lu	
É		311	201	C9	Latin Capital Letter E With Acute		Lu	
Ê		312	202	CA	Latin Capital Letter E With Circumflex		Lu	
Ë		313	203	CB	Latin Capital Letter E With Diaeresis		Lu	
Ì		314	204	CC	Latin Capital Letter I With Grave		Lu	
Í		315	205	CD	Latin Capital Letter I With Acute		Lu	
Î		316	206	CE	Latin Capital Letter I With Circumflex		Lu	
Ï		317	207	CF	Latin Capital Letter I With Diaeresis		Lu	
Ð		320	208	D0	Latin Capital Letter Eth		Lu	
Ñ		321	209	D1	Latin Capital Letter N With Tilde		Lu	
Ò		322	210	D2	Latin Capital Letter O With Grave		Lu	
Ó		323	211	D3	Latin Capital Letter O With Acute		Lu	
Ô		324	212	D4	Latin Capital Letter O With Circumflex		Lu	
Õ		325	213	D5	Latin Capital Letter O With Tilde		Lu	
Ö		326	214	D6	Latin Capital Letter O With Diaeresis		Lu	
×		327	215	D7	Multiplication Sign		Sm	
Ø		330	216	D8	Latin Capital Letter O With Stroke		Lu	
Ù		331	217	D9	Latin Capital Letter U With Grave		Lu	
Ú		332	218	DA	Latin Capital Letter U With Acute		Lu	
Û		333	219	DB	Latin Capital Letter U With Circumflex		Lu	

Char-acter	Also Called	Octal Value	Decimal Value	Hex Value	ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name	Also Known As	Type.	See Note
Ü		334	220	DC	Latin Capital Letter U With Diaeresis		Lu	
Ý		335	221	DD	Latin Capital Letter Y With Acute		Lu	
Þ		336	222	DE	Latin Capital Letter Thorn		Lu	
ß		337	223	DF	Latin Small Letter Sharp S		Ll	
à		340	224	E0	Latin Small Letter A With Grave		Ll	
á		341	225	E1	Latin Small Letter A With Acute		Ll	
â		342	226	E2	Latin Small Letter A With Circumflex		Ll	
ã		343	227	E3	Latin Small Letter A With Tilde		Ll	
ä		344	228	E4	Latin Small Letter A With Diaeresis		Ll	
å		345	229	E5	Latin Small Letter A With Ring Above		Ll	
æ		346	230	E6	Latin Small Ligature AE		Ll	2
ç		347	231	E7	Latin Small Letter C With Cedilla		Ll	
è		350	232	E8	Latin Small Letter E With Grave		Ll	
é		351	233	E9	Latin Small Letter E With Acute		Ll	
ê		352	234	EA	Latin Small Letter E With Circumflex		Ll	
ë		353	235	EB	Latin Small Letter E With Diaeresis		Ll	
ì		354	236	EC	Latin Small Letter I With Grave		Ll	
í		355	237	ED	Latin Small Letter I With Acute		Ll	
î		356	238	EE	Latin Small Letter I With Circumflex		Ll	
ï		357	239	EF	Latin Small Letter I With Diaeresis		Ll	
ð		360	240	F0	Latin Small Letter Eth		Ll	
ñ		361	241	F1	Latin Small Letter N With Tilde		Ll	
ò		362	242	F2	Latin Small Letter O With Grave		Ll	
ó		363	243	F3	Latin Small Letter O With Acute		Ll	
ô		364	244	F4	Latin Small Letter O With Circumflex		Ll	
õ		365	245	F5	Latin Small Letter O With Tilde		Ll	
ö		366	246	F6	Latin Small Letter O With Diaeresis		Ll	
÷		367	247	F7	Division Sign		Sm	
ø		370	248	F8	Latin Small Letter O With Stroke		Ll	
ù		371	249	F9	Latin Small Letter U With Grave		Ll	
ú		372	250	FA	Latin Small Letter U With		Ll	

Char-acter	Also Called	Octal Value	Decimal Value	Hex Value	ISO/IEC 10646-1:1993(E) and Unicode 2.0 Name	Also Known As	Type.	See Note
					Acute			
û		373	251	FB	Latin Small Letter U With Circumflex		Ll	
ü		374	252	FC	Latin Small Letter U With Diaeresis		Ll	
ý		375	253	FD	Latin Small Letter Y With Acute		Ll	
þ		376	254	FE	Latin Small Letter Thorn		Ll	
ÿ		377	255	FF	Latin Small Letter Y with Diaeresis		Ll	

Notes:

1. This code point is used by National Replacement Character Sets (7-bit character sets). Devices using such an NRC will not print the glyph shown, neither will it print glyphs for code points above 127.

2. ISO may be reclassifying these code points as “Latin Letter”, as certain Scandinavian languages use these characters as a complete letter, not as a ligature.

Type: The characters are broken down into “character types” by Unicode:

Cc	Control or Format Character
Ll	Lowercase Letter
Lm	Modifier Letter
Lu	Uppercase Letter
Nd	Decimal Number
Pd	Dash Punctuation
Pe	Close Punctuation
Po	Other Punctuation
Ps	Open Punctuation
Sc	Currency Symbol
Sm	Math Symbol
So	Other Symbol
Zs	Space Separator

Appendix G
RAD50 Conversion Table

For historical reasons, AMOS often stores and retrieves text strings as packed RAD50 data. (For example, filenames within disk directories are stored as RAD50 data.) ASCII characters take up one byte each. Using the RAD50 packing algorithm, you can "pack" three ASCII characters into two bytes.

The ASCII characters that you may pack into RAD50 data are:

Character	RAD50 Code (Octal)
blank	0
A-Z	1-32
a-z	1-32
$	33
.	34
%	35
0-9	36-47

NOTE: the codes on the right are used to compute the RAD50 values of the characters.

The chart below gives you a method for manually converting RAD50 data back into its ASCII character equivalents, and also for converting ASCII characters into RAD50 data.

THE RAD50 ALGORITHM

If you are interested in the actual algorithm used to pack three ASCII characters into two RAD50 bytes, here is a summary:

1. Multiply the first character code by 3100_8. (To find out the character's code, consult the short table above.) (3100_8 is 50_8 x 50_8—hence, the name RAD50.)
2. Multiply the second character code by 50_8 and add it to the first number.
3. Add the third character code to the above. This final sum is the RAD50 result.

To convert RAD50 data back into ASCII characters, reverse the sequence above, subtracting instead of adding.

Note also that the PACK and UNPACK monitor calls are available for your use—these monitor calls convert ASCII characters into RAD50 data and vice versa. (Consult Chapter 8, *Conversion Monitor Calls*, for information on these monitor calls.)

USING THE CONVERSION CHART

To convert ASCII characters into RAD50:

1. Look in the first column on the left for the first ASCII character you want to pack. Write down the octal number associated with it.
2. Look in the second column for the second ASCII character you want to pack. Write down the octal number associated with it beneath the first number you wrote down. Now, add these numbers. (Remember, add them in octal.)
3. Look in the third column for the third ASCII character you want to pack. Write down the octal number associated with it beneath the sum of the first two numbers. Add in the new number.

The final number is the 16-bit RAD50 result.

To convert RAD50 data into ASCII characters:

1. Write down the RAD50 octal data you want to convert.
2. Look in the first column on the left for the largest octal number that can be subtracted from your RAD50 data without producing a negative number. The character associated with that number is the first ASCII character. Now, subtract the octal number from the RAD50 data.
3. Look in the second column for the largest octal number that can be subtracted from your new result without producing a negative number. The character in the second column that is associated with that number is the second ASCII character. Subtract the octal number from the current RAD50 result.
4. Look in the third column for the largest octal number that can be subtracted from your new result without producing a negative number. The character in the third column associated with that number is the final ASCII character.

For example:

To convert the ASCII characters "DSK" into RAD50:

1. Look for "D" in the first column. Write down its associated number:

```
 014400
```

2. Look for "S" in the second column. Write down its associated number, and add it to the data above:

```
 014400
+001370
 -----------
 015770
```

3. Look for "K" in the third column. Write down its associated number and add it to the result above:

```
 014400
+001370
 -----------
 015770
+000013
 -----------
 016003
```

The octal, 16-bit RAD50 representation of the ASCII characters "DSK" is 016003.

To convert the RAD50 data "016003" into three ASCII characters:

1. Look for the largest number in the first column that you can subtract from 016003 without producing a negative number (014400). The character associated with 014400 is "D." Subtract the number:

```
 016003
-014400
 -----------
 001403
```

2. Look for the largest number in the second column that you can subtract from 001403 without causing a negative result (001370). The character associated with 001370 is "S." Subtract the number:

```
 016003
-014400
 -----------
 001403
-001370
 -----------
 000013
```

3. Look for the result above in the third column (000013). The character associated with that number is "K."

The RAD50 number 016003 represents the three ASCII characters "DSK."

THE CONVERSION CHART

CHARACTER #1		CHARACTER #2		CHARACTER #3	
Blank	000000	Blank	000000	Blank	000000
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000016
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
$	124300	$	002070	$	000033
.	127400	.	002140	.	000034
%	132500	%	002210	%	000035
0	135600	0	002260	0	000036
1	140700	1	002330	1	000037
2	144000	2	002400	2	000040
3	147100	3	002450	3	000041
4	152200	4	002520	4	000042
5	155300	5	002570	5	000043
6	160400	6	002640	6	000044
7	163500	7	002710	7	000045
8	166600	8	002760	8	000046
9	171700	9	003030	9	000047

Appendix H
User Description Symbols

The symbols below can be used to access user description information contained in the DSK0:USER.SYS[1,2] file via the $FNUSR library call discussed in Appendix D. In addition to the symbols below, US.SIZ contains the total number of bytes used by the US.xxx symbols.

Symbol	Size (Bytes)	Description
US.USN	20.	User name (ASCIZ)
US.FLG	4.	User flags (low word matches JOBTYP)

Symbol	Octal Value	Hex Value	Meaning
US$HEX	20	10	Default radix is hex
US$DER	40	20	Disk error print
US$VER	100	40	Disk verify
US$CCA	200	80	Control-C enable
US$GRD	400	100	Guard
US$NLK	100000	8000	No LOKSER locking on traditional logicals
US$ENLK	200000	10000	No LOKSER locking on extended logicals
US$ECH	20000000000	80000000	Echo suppress

Symbol	Size (Bytes)	Description
US.RTP	2.	Root PPN
US.RTD	2.	Root device
US.RTU	2.	Root unit
US.RTC	4.	Root CPU
US.MLP	2.	Mail PPN
US.MLD	2.	Mail device
US.MLU	2.	Mail unit
US.MLC	4.	Mail CPU
US.PRV	2.	Default privileges
US.CLS	1.	Class of user
US.EXP	1.	Experience level of user
US.PAS	10.	Password
US.CPU	4.	Total CPU time
US.CON	4.	Total connect time
US.KRM	4.	Kilo-RAM-seconds used
US.DRD	4.	Total disk reads
US.DWR	4.	Total disk writes
US.PRT	4.	Total pages printed
US.DFP	4.	Default file protection
US.PRI	2.	Default job priority
US.LNG	20.	Default language
US.PRM	20.	Default prompt

Not all fields are currently in use or updated by AMOS.

Appendix I
Eight-Bit Character Support

With the addition of support for full 8-bit character sets (such as ISO Latin-1) to AMOS, programs capable of handling a full 8-bit character set are able to request that all characters are sent and received without modification. Function key handling has also been updated to support full 8-bit transparency.

AMOS has traditionally supported the 7-bit ASCII character set. Languages other than American English have been supported through the use of National Replacement Character (NRC) sets, whereby some of the symbols of USASCII are replaced by other symbols required by the local language.

In practice, this has been only a stopgap solution. Many applications are hindered by the lack of the symbols used for substitution. Multilingual users have not been able to mix character sets. These and many other limitations have led to the need for a better solution.

To expand the number of available characters, AMOS is moving to support a full 8-bit character set. AMOS has traditionally removed the high-order bit of all ASCII characters, usually using it as a parity bit (applications software often uses the high-order bit to "hide" flags associated with the characters). In addition, the high-order bit has been used by AMOS to flag function key input. This makes the transition to 8-bit character sets more difficult.

As of AMOS 2.2, AMOS has an 8-bit transparent mode, functional in line, image, and data input modes. Appropriate changes were made within AMOS to allow this. Applications software, however, needs to be upgraded to take full advantage of this new capability.

Although AMOS assumes the use of the ISO Latin-1 character set to the extent that the text files distributed with AMOS will use that set, the 8-bit transparency feature does not limit the choice of specific character sets.

Note also that AMOS still assumes a single byte per character, although the new 8-bit feature is designed not to hinder users of multi-byte character sets.

Here is a brief summary of the changes made for 8-bit character support:

- A program can check for the availability of 8-bit character set support within AMOS by checking for SY$EXT being set in the system flags (SYSTEM in the system communication area).
- A program can set the T$EXT bit in its terminal status word to receive full 8-bit character input.
- Terminal drivers flag their ability to generate and display 8-bit characters by setting the TD$EXT bit in the terminal driver flags word, retrieved by the TRMCHR monitor call.
- When in 8-bit character mode, function keys are returned as two-byte sequences: a hex 9B (Control Sequence Introducer— CSI) is a special ISO character that is followed by the same function key byte returned by previous versions of AMOS.

- A program that wants to accept 8-bit characters can set the PH$EXT bit in the program header flags, thus eliminating the need to explicitly set T$EXT by TRMRST/TRMWST. This also allows for easy upgrading of programs that already support 8-bit character sets.
- Programs that set the high-order bit in an attempt to get the character sent "transparently" must be changed to not do this in order to work with 8-bit characters. Common examples of where changes are needed are programs that set the high-order bit on carriage returns (to avoid the line feed being appended), tabs (to avoid the expansion to spaces) and bell (for no apparent reason).

8-bit character support in AMOS has been limited to the AMOS monitor itself, as well as some key utilities, such as AlphaVUE and AlphaBASIC. Some programs shipped with AMOS may still be limited to 7-bit ASCII.

The AMOS SORT program and the AlphaBASIC XCALL BASORT have been changed to sort according to the type of character set currently being used. The language definition file format has been updated to include additional fields required for 8-bit support.

COMPATIBILITY

To facilitate upgrading from software which only recognizes 7-bit character sets, AMOS allows a mix of software on the system. Full 8-bit characters are sent to those programs which know how to deal with them, while the set is reduced for 7-bit programs.

Both terminals and programs can be either 7- or 8-bit. AMOS handles all four combinations of these. When a 7-bit terminal uses an 8-bit program, function key input is converted from single byte to two-byte codes. It is up to the application to convert characters between the terminal's NRC and the program's 8-bit code.

When an 8-bit terminal uses a 7-bit program, function key input is sent to the program as single byte codes. It is up to the application to convert characters between the terminal's 8-bit code and the program's NRC code.

The conversion between NRC and 8-bit should be done by a conversion table. These tables should be both terminal-dependent and language-dependent. They should have the name term.lan, where term is the name of the terminal, and lan is the language dependent extension from the currently selected language definition table. By making the table dependent on both terminal type and language, it is possible to handle the various differences in NRC sets. These table files actually consist of two tables. The first 128-byte table, when indexed with the 7-bit NRC character, will give the 8-bit equivalent. A value of zero means there is no equivalent. The second 256-byte table, when indexed with an 8-bit character value, will give the 7-bit NRC equivalent. Again, a null means there is no equivalent.

UPGRADING AN APPLICATION

For all applications:

- All use of the high-order bit must be removed. This includes use inside of buffers, and any masking of the high-order bit currently being done.

- The program must set the T$EXT flag either through explicit use of TRMRST/TRMWST or by setting the PH$EXT flag in the program header.
- Determine if the terminal in use is 7- or 8-bit and, when 7-bit, make appropriate use of translation tables to convert back and forth between NRC and 8-bit characters.
- Make appropriate adjustments to comparison and sort routines to not sort based on simple ASCII value, but on the collating table in the language definition file.

In addition, programs which run in data or image mode with function keys need to update the function key handler to recognize the function key lead-in of hex 9B, discard that, and use the following byte as the function code.

Upgrading a Terminal Driver

Terminals which only support 7-bit ASCII do not need any modifications. Terminals that are capable of supporting an 8-bit character set need to have their drivers modified in two areas:

- The TD$EXT bit needs to be set in the terminal flags word.
- The function key handler needs to be changed—when the terminal is in 8-bit mode, function keys are returned as two-byte sequences. When T$EXT is not set, the TDV must return a single byte code.

Not all terminals available on the Alpha Micro computer may support 8-bit characters. See your dealer for information on what terminals (and terminal drivers) are available that support 8-bit characters. Remember that the terminal itself must be set to generate the full 8-bit ISO character set.

Appendix J
Using AMSORT.SYS

AMSORT.SYS is a sort module included with AMOS which can be easily called from assembler or C programs. It uses a binary insertion sort method, which yields a stable sorting sequence. This means that two different records will be returned in the same order as they were presented to AMSORT if their sort keys compare equally. AMSORT can transparently handle more data than can fit in memory, automatically paging data in and out of memory as needed without extra work on the programmer's part. The number of keys supported is limited only by the amount of free memory. AMSORT calls programmer-defined routines to read and write data, providing complete flexibility for data sources and sinks.

AMSORT can deal with the following key data types:

- 7-bit ASCII strings. It can optionally support 8-bit characters and the extended collating sequence defined in the job's language definition file. Character by character comparisons are done, proceeding from lower locations to higher ones. If the optional extended collating flag is set, the key is translated to and from the collating sequence characters to perform the comparison.
- AMOS floating point numbers. AMSORT uses the FCMP call to compare the keys.
- Binary data. AMSORT uses integer data in AlphaBASIC unsigned binary data format.

Non-key data in a record can be in any format.

AMSORT is easy to use. The following steps give an overview of the process:

1. Load AMSORT into user memory if it is not already loaded in user or system memory.
2. Use the AMCHK macro to verify that AMSORT.SYS is a current version.
3. Point A5 at an impure area. Details about this impure area are given below.
4. Check the active language for the job, and decide whether or not to call the 8-bit character collation initialization sequence in AMSORT. If so, set the COLL8 flag using the XTDINI macro.
5. Define the area in memory that can be used by AMSORT for sorting purposes. This area is separate from, but can be contiguous with, AMSORT's impure area.
6. Set up the key list in the impure area using the KEYOFF macro.
7. Clear any unwanted features flags in the impure area using the FLGCLR macro.
8. Set the maximum record size of the records being sorted in the impure area.
9. Set the addresses of your own input and output routines in the impure area.
10. Call AMSORT.

The macros mentioned above are defined in the AMSYM.UNV file, which should be SEARCHed in your source file.

LOCATING AND VERIFYING AMSORT.SYS

AMSORT can be located in memory by using the SRCH and FETCH monitor calls. The AMCHK macro returns with the Z flag set if the AMSORT.SYS module is usable.

THE IMPURE AREA

AMSORT uses an impure area which must be indexed by A5 for variable storage and the key list table. The variable storage is an area of fixed length that holds variables you set, as well as internal variables private to AMSORT. You are responsible for building the key list table at the end of the variable storage area with the help of the KEYOFF macro. The size of the impure area which must be allocated is calculated by adding the minimum impure area size to N * KEYSIZ, where N is the number of sort keys to be defined, and KEYSIZ is the size of a key table entry, currently 14. bytes. To find the minimum impure area size, locate AMSORT.SYS in memory, and use:

```
KEYOFF   An, Dn
```

An is an address register holding the address of the AMSORT.SYS module in memory, and ***Dn*** receives the impure offset value to the key list table. Dn also represents the minimum impure area size. Add N * KEYSIZ to Dn, and point register A5 to an area of that size or larger to define the impure area. The start of the key table list, Am, is then given by:

```
LEA      Am, 0(A5)[Dn]
```

where ***A5*** indexes the impure area. Each key table entry is defined as follows:

Entry	Size	Description
SIZE	longword	Length of key in bytes, maximum = 65,535 bytes
LOCATION	longword	Offset into the record for this key. The first byte of the record is byte zero.
SPARE	longword	Reserved, set to null
DIRECTION	byte	Sorting direction: bit 7 set Ascending bit 7 clear Descending All other bits Reserved, set to zero.
TYPE	byte	Type of key: 0 String 1 AMOS 6-byte floating point 2 AlphaBASIC binary
KEYSIZ = 14.		Size of a key table entry

The list is terminated by a key table entry with a SIZE set to zero. The key table list must be set up before AMSORT is called.

Within the impure area are two variables: SM.FRE(A5), which defines the start of the contiguous area which AMSORT can use for sorting; and SM.END(A5), which defines the address of the last word of the area. Both addresses must be even addresses. AMSORT will longword align these two values, so they can change after AMSORT is called. The maximum size of this area is 32KB. If a larger block is defined, AMSORT will shrink the area during initialization by altering the SM.END(A5) value.

SETTING THE RECORD SIZE

The maximum size of a record is set into the longword at SM.REC(A5). Only the lower word is used, providing a maximum record size of 65,535 bytes. The total of each key's SIZE and LOCATION must be less than or equal to SM.REC(A5).

SETTING FLAGS

The FLGCLR macro is then called to initialize the flags in AMSORT's impure area.

If you do not want AMSORT to use the job's language definition file's extended 8-bit collating sequence, clear the byte at COLL8(A5). If you want the extended collating sequence used, place a non-zero byte in COLL8(A5), and initialize 8-bit processing by calling the XTDINI macro. If XTDINI is not called when it should be, or is called when the LDF does not contain an extended collating sequence, the sorted records returned will be corrupted. XTDINI needs to be called only when the first (or a new) impure area for AMSORT is being defined.

INPUT AND OUTPUT ROUTINES

You define your input and output routines to AMSORT by placing their addresses in SM.IN(A5) and SM.OUT(A5) respectively. Any registers used in your routines must be preserved by those routines. AMSORT will keep calling your input routine until you return a zero in register D0. You must place exactly SM.REC(A5) bytes of data (padded with nulls if needed) into the area indexed by A1. Return non-zero in register D0 and a zero in D1 until the last record is being processed, then return the last record with registers D0 and D1 set to zero. If your input routine is called again, immediately return a -1 in register D1 to signal an error condition.

After you have sent the last record to AMSORT, AMSORT will start calling your output routine with register A1 pointing to the record you are to receive. Register D2 will be set to SM.REC(A5) to help you do the transfer from AMSORT's buffer to your data sink. After AMSORT has processed all the records, it will call your output routine again with A1 cleared to zero. On return from that call, AMSORT will clean up after itself, and return to the calling program.

EXAMPLE PROGRAM

```
        SEARCH  SYS
        SEARCH  SYSSYM
        SEARCH  AMSYM

NITEMS  =       9.            ;number of data items
RECSIZ  =       5             ; record size
NKEYS   =       1             ; number of keys

        PHDR    -1,,          ; This program is not reentrant or reusuable.
                              ; Initializing COUNT to zero at the start
                              ; would make it reusable, and making
                              ; OUTLST be in another impure area
                              ; would make it reentrant.
        SRCH    AMNAME, A2    ; look for AMSORT.SYS in memory
        BEQ     10$           ; ....br if found....
        TYPECR  <?AMSORT.SYS not in memory!> ; ....else error, abort
        EXIT
```

```
10$:    AMCHK   A2              ; version check
        BEQ     20$
        TYPECR  <?AMSORT.SYS is not current, and cannot be used>
        EXIT

20$:    KEYOFF  A2, D6                   ; D6 holds min impure size
        ADD     #KEYSIZ*NKEYS, D6        ; add key table size
        ADD     #4, D6                   ; allow for terminating null
        PUSH    D6
        PUSH
        GETMEM  @SP                      ; get module for impure area
        BEQ     30$
        TYPECR  <?Insufficient memory for sorting!>
        EXIT

30$:    POP     A5                ; A5 set to impure area
        POP                       ; stack restored
        GETIMP  1024., A4         ; allocate 1kbyte of sorting space
        MOV     A4, SM.FRE(A5)    ; set in start of sort space into impure area
        ADD     #1024., A4        ; set upper bound of block....
        MOV     A4, SM.FRE(A5)    ; ....into impure area

        LEA     A3, INSBR         ; a3 -> my input routine
        MOV     A3, SM.IN(A5)     ; ....set into impure area
        LEA     A3, OUTSBR        ; a3-> my output routine
        MOV     A3, SM.OUT(A5)    ; ....set into impure area

        MOV     #RECSIZ, SM.REC(A5)     ; test data record is 5 bytes long
                                  ; (includes terminating null in this case)

        FLGCLR  A5              ; clear flags

        CLRB    COLL8(A5)       ; default to ignore 8-bit collating
        MOV     SYSTEM, D7      ; D7 holds SYSTEM lword
        AND     #SY$LNG, D7     ; is .LDF file supported????
        BEQ     40$             ; ....no, use default, br
        GTLANG  A3
        CMPW    LD.XTV(A3), #^H05A5A   ; extended .LDF present????
        BNE     40$             ; ....no, use default, br
        MOVB    #1, COLL8(A5) ; set up for extended collating
        XTDINI  A2              ; initialize...only if extended collating

40$:    KEYOFF  A2, D6          ; d6 holds key table offset address
        LEA     A3, 0(A5)[D6] ; A3 indexes the area
        MOV     #4, (A3)+       ; key is four bytes long....
        CLR     (A3)+           ; ....at byte zero in record....
        CLR     (A3)+           ; reserved, must be zero
        CLRB    (A3)+           ; ascending order
        CLRB    (A3)+           ; string type
        CLR     (A3)+           ; end of list

        CALL    @A2             ; call AMSORT!

        TYPECR  <The sorted list:>
        MOV     #NITEMS-1, D3 ; d3 holds items less DBF adjustment
        LEA     A3, OUTLST      ; a3 -> list
50$:    TTYL    @A3             ; output item
        CRLF
        ADD     #RECSIZ, A3     ; a3 -> next item
        DBF     D3, 50$         ; loop until done
EXIT

INSBR:  PUSH    A3              ; save registers I use
        LEA     A3, COUNT       ; a3 -> items processed
        CLR     D0              ; preclear
        MOVB    @A3, D0         ; get count
        CMPB    D0, #NITEMS     ; if all processed....
        BLOS    10$
        MOV     #-1, D1         ; ....error, signal AMSORT, br
        BR      40$
```

```
10$:    MOV     D0, D1          ; save count
        INC     D0              ; inc count here....
        INCB    @A3             ; ....and in memory
        CMPB    D0, #NITEMS     ; all done????
        BLO     20$             ; ....no, br
        CLR     D0              ; yes, so signal AMSORT as well
20$:    LEA     A3, SRTLST      ; a3 -> buffer area
        MUL     D1, #5          ; d1 converted to offset
        ADD     D1, A3          ; a3 -> area to hold item
30$:    MOVB    (A3)+, (A1)+    ; move from our buffer to AMSORT’s
        BNE     30$             ; ....until done
        CLR     D1              ; signal "no error"
40$:    POP     A3              ; restore registers
        RTN                     ; return to AMSORT

OUTSBR:
        SAVE    A3, D0, D1      ; save registers
        LEA     A3, COUNT       ; a3 -> count
        CLR     D0              ; preclear
        MOVB    @A3, D0         ; get count of items to be processed
        BEQ     20$             ; if all done, br
        DECB    @A3             ; dec count
        MOV     #NITEMS, D1     ; d1 holds max # items
        SUB     D0, D1          ; invert about that number
        MUL     D1, #5.         ; convert item number into offset
        LEA     A3, OUTLST      ; a3 -> buffer area
        ADD     D1, A3          ; a3 -> place to put item
10$:    MOVB    (A1)+, (A3)+    ; move from AMSORT's buffer to ours
        BNE     10$             ; ....until done
20$:    REST    A3, D0, D1      ; restore used registers
        RTN                     ; return to AMSORT

AMNAME:
        RAD50   /AMSORTSYS/
COUNT:
        BYTE    0

SRTLST:
        ASCIZ   /GGGG/
        ASCIZ   /BBBB/
        ASCIZ   /BBBC/
        ASCIZ   /1234/
        ASCIZ   / SPC
        ASCIZ   /.PER/
        ASCIZ   /AAAA/
        ASCIZ   /aa  /
        ASCIZ   /aaa /

OUTLST:
        BLKB    NITEMS * RECSIZ

        END
```

Document History

Revision	Release	Date	Description
A00	AMOS/L 1.0	4/82	New Document, Part Number DSS-10003-00.
A01	AMOS/L 1.1	3/83	Added LOKSER Monitor Calls to Chapter 6, added LOKSER fields in the system communication area to Appendix C, and updated alphabetic listing of monitor calls in Appendix E.
A02	AMOS/L 1.2	5/84	Added TMRLOK, JOBFPC, JOBERC, and TRMCHR calls. System flags SY$LNK, SY$LOK, SY$HFP added. Changes were made to the TIMER call. The TTYOUT call was implemented. New TCRT codes (79-99) and support codes for color terminals were added. Routines $BBCHK and $ERPPN were added, and routine $INFMD was renamed to $INMFD. Some small errors were also corrected.
A03	AMOS/L 1.3	6/85	Added JOBLNG, JOBUSN, JOBRTP, JOBRTD, JOBRTU, JOBLVL, JOBEXP, JOBPRM, and JOBCMD calls to the JCB. Added to the FSPEC call. Added the T$XLT and T$NFK flags to the terminal status word. Added TCRT functions 100-147, and reserved all others for Alpha Micro use only. Added the OT$NLD flag to the FCVT call. Added the DEVCHR call. Added chapters 14, 15, and 16. Added new status flags to the SYSTEM attributes word, and the ERSATZ, SYSNAM, and SYSLNG calls to the system communication area. New system library routines include $KILPF, $NETED, $SPLFL, $STFRM, $SYSID, $UPDSW, and $YESNO. Generally, support for the user-definable ersatz names, function key translation, disk cache buffer manager, intertask communication and foreign language/central message support.
Rev. 00	AMOS 2.0	3/88	Rewritten and assigned new part number for AMOS 2.0 release, DSO-00040-00. Added many new calls to handle expanded AMOS capabilities and extended disk format.
Rev. 01	AMOS 2.0A	12/88	Minor corrections: new status bits for the TAPST magnetic tape status call, clarification of the use of the AMOS and DQTIMR calls, information on the $FNUSR library routine, information on the new JLOCKI call, updated information in Chapter 16, “Serial Communications System,” Appendix H, “User Description Symbols.”

(Continued)

Revision	Release	Date	Description
Rev. 02	AMOS 2.1	9/89	Minor typographical corrections; new J2$BGT flag for JOBTY2; new queue management calls QADDL, QINSL, QUNL; new GETX, INPUTX, and CLOSEK calls for file system; new fields added to system communication area; new system library routine $DITOS; additions to $SPLFL library routine.
Rev. 03	AMOS 2.2	4/91	New system attribute word flags added for identifying new computer systems; eight-bit character support documented; miscellaneous terminal status word flags are documented in Chapter 7 and Appendix B; new error codes for $SPLFL library routine added; new system library routines $MSGLOG and $CPUPOL added; additions to Software Interrupt System; file locking changes adding new flags; floating point clarifications and new call FFTOAX (Floating Point to ASCII Extended) added; new shared memory facility documented.
Rev. 04	AMOS 2.3	9/96	Changes and additions to Appendix D. Added $..PPNX and $..MFDX routines and changed $..PPN and $..MFD routines. Changed $CPUPOL and $SPLFL. Year 2000 support: changed Chapter 10, revised $ODTIM, $PAKDT, $UNPDT, added $IDTIMX and $ODTM2. Chapter 2: added job priority and interpreted prompt information. Added LEVEL7, ICON, ICOFF to Chapter 13. Other miscellaneous changes to chapters 2, 6, 15, and 16. New System Communications Area words in Appendix C. All new Appendix F: ISO Latin-1 character set.
Rev. 05	AMOS 2.3A	5/97	Revised $ODTIM in Appendix D.
Rev. 06	AMOS 2.3A	12/97	Added Chapter 20, “AlphaTCP Programming Interface” and Appendix J, “Using AMSORT.SYS.” Reformatted and edited entire document.
Rev. 07	AMOS 2.3A	5/98	Correct error in $IDTIMX description in Appendix D.
Rev. 08	AMOS 2.3A, PR 6/98	6/98	Change date range for $DITOS, $DSTOI, $IDTIM and $IDTIMX; add bit 2 flag to $IDTIM and $IDTIMX, update $ODTM2; add flags in Appendix H.
Rev 09	AMOS 2.3A, PR 10/99	10/99	Added SYNC and SPAWN in Chapter 13 and Appendix E.
Rev 10	AMOS 2.3A, AlphaTCP 1.5A	4/00	Added GETVTI in Chapter 13. Added Hostname and Address Conversion call TCPRES

Index